LUC STEELS

programmeren in LISP

2e druk

ACADEMIC SERVICE



40.00

Programmeren in LISP

PROGRAMMEREN IN LISP

Luc Steels

ACADEMIC SERVICE

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Steels, Luc

Programmeren in LISP / Luc Steels. - Den Haag : Academic Service
ISBN 90-6233-229-3
SISO 365.3 SVS 8.12.3 UDC 681.3.06 UGI 200
Trefw.: LISP (programmeertaal).

1e druk 1983
2e geheel herziene druk 1986

Uitgegeven door: Academic Service
Postbus 81
2870 AB Schoonhoven
Omslagontwerp: JAM Gauw
Zetwerk: door de auteur gezet in Times Roman met behulp van TROFF, een onder UNIX draaiend tekstverwerkend systeem
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam
ISBN 90 6233 229 3

"Bij wijze van grap heeft iemand LISP ooit de intelligentste manier genoemd om een computer te misbruiken. Ik vind deze beschrijving een groot compliment omdat het goed weergeeft dat LISP een echte bevrijding is: Het heeft enkele zeer talentvolle mensen geholpen om gedachten te hebben die daarvoor onmogelijk waren."

E. Dijkstra, *The Humble Programmer*, Comm. ACM, 15, 1972.

VOORWOORD

Dit boek is bedoeld voor informatici die een basis in LISP willen verwerven. LISP is een programmeertaal met enkele heel bijzondere eigenschappen:

1. Het is een interactieve taal: op elk moment kan een stuk programma toegevoegd of veranderd worden zonder de rest opnieuw te moeten inlezen of compileren en op elk moment kan het evaluatieproces gestopt en weer voortgezet worden, eventueel na wijzigingen aan het programma of interne datastructuren. Dit maakt het veel gemakkelijker om fouten op te sporen en te verbeteren of om programma's aan te passen.

2. LISP heeft krachtige primitieve datastructuren voor symbolisch programmeren: atomen en lijsten. Vooral lijsten zijn bijzonder handig om andere dynamische datastructuren op te bouwen. Bovendien moet de programmeur zich niet bezighouden met het organiseren of onderhouden van het geheugen zodat hij op een zeer flexibele wijze met deze structuren kan werken. LISP is daarom erg geschikt voor symbolisch programmeren. De taal is echter evenwaardig aan andere talen wat betreft numerieke capaciteit en is uitstekend geschikt voor systeemprogrammering.

3. Programma en data hebben dezelfde structuur in LISP en het mechanisme dat LISP-programma's uitvoert (de evaluator) is zelf een LISP-functie die expliciet kan worden opgeroepen. Dit maakt het gemakkelijk om de taal uit te breiden met nieuwe data- of controle-structuren en om een stuk programma als datum te behandelen. LISP is daarom uiterst geschikt om te experimenteren met nieuwe programmeermechanismen en programmeermethoden.

4. Het is daarom ook mogelijk om diverse programmeerstijlen in LISP te gebruiken. We zullen voorbeelden bestuderen van een applicatieve, een functionele, een objectgerichte en een imperatieve programmeerstijl. Elk van deze

stijlen heeft zijn voordelen voor bepaalde toepassingen en LISP laat toe om te kiezen voor een bepaalde methode zonder te moeten veranderen van programmeertaal.

5. LISP moedigt modulariteit en gestructureerde programma's aan. Bijvoorbeeld in een puur applicatief gebruik van LISP kunnen functies samengesteld worden zonder gevaar voor subtiele interacties. Er zijn geen beperkingen op de argumenten of het resultaat van een functie, zodat gelijk welk type van object kan gedefinieerd en gebruikt worden als een coherente datastructuur.

6. LISP is de standaardtaal voor onderzoek in kunstmatige intelligentie en is belangrijk in de theoretische informatica omwille van haar logische fundering en elegante wiskundige eigenschappen.

Er zijn dus redenen genoeg om LISP te leren en de taal vormt terecht een onderdeel van de opleiding tot informaticus.

Een taal kan men het best leren door haar te gebruiken. Daarom bevat dit boek een groot aantal oefeningen. Sommige hiervan zijn vrij eenvoudig en bedoeld om te zien of de tekst werd begrepen. Andere zijn moeilijker of van grotere omvang. De oefeningen vormen een belangrijk onderdeel van de tekst. Het is de bedoeling dat de lezer ze doorneemt alvorens verder te gaan. Lectuur en oefeningen moeten uiteraard aangevuld worden met computerpraktijk.

Er is geen standaard LISP maar het is in principe geen groot probleem om van het ene systeem naar het andere over te stappen. Dit boek gebruikt een variant van de MacLISP-familie. Documentatie van het locale LISP systeem moet worden geraadpleegd voor dialect-specifieke details, en ook voor input/output, editor, compilator, enz.

Het eerste deel introduceert een basisrepertorium van primitieve elementen en functies. Het tweede deel behandelt het mechanisme voor de definitie van een functie en enkele belangrijke definitieschema's, voornamelijk van het recursieve type. Het derde deel handelt over getallen, strings en imperatief programmeren, het vierde deel over functies van hogere orde en functioneel programmeren en het vijfde deel over structuren in LISP en objectgericht programmeren. Het zesde deel gaat nader in op de evaluatie van formules en hoe een LISP evaluator geschreven kan worden in LISP. Het laatste deel beschrijft de scala van LISP dialecten en LISP machines. Achteraan bevindt zich een geannoteerde bibliografie en een index.

Dit boek kan dienen als handboek bij een cursus over LISP of voor zelfstudie. Na lectuur moet de lezer in staat zijn om LISP-programma's te schrijven, zelfs voor moeilijke problemen, en om programma's van anderen te lezen en te begrijpen.

Voorwoord

Alhoewel men ook imperatief kan programmeren in LISP, wordt hoofdzakelijk applicatief programmeren bestudeerd, d.w.z. puur LISP. Belangrijke basisconcepten van de informatica worden geïllustreerd aan de hand van LISP voorbeelden en er wordt ingegaan op de werking van LISP. Er zijn uiteraard heel wat onderwerpen die in een basistekst niet aan de orde kunnen komen, bijvoorbeeld niet-locale controlestructuren, closures, implementatie, enz. De bibliografie bevat enkele werken die hierop verder ingaan.

Dit boek is gegroeid uit colleges aan de universiteit van Brussel. Kritisch commentaar van studenten heeft een belangrijke bijdrage geleverd. De auteur bedankt ook Ir. J. Van der Hijden en de uitgever drs. H.J. Stomps voor heel wat constructieve opmerkingen en Dora De Haeck voor logistieke en linguïstieke steun.

Luc Steels
Parijs, November 1982.

VOORWOORD BIJ TWEEDE DRUK

Sinds het verschijnen van dit boek zijn er een aantal belangrijke dingen gebeurt die een tweede editie zinvol maken. Ten eerste is de AI die één van de belangrijkste gebruikers blijft van LISP volop in de belangstelling komen te staan. Hierdoor is de behoefte om LISP te kennen enorm toegenomen.

Ten tweede is LISP beschikbaar geworden op zeer veel machines. Niet alleen zijn de LISP machines één van de grote commerciële successen geworden van de AI technologie. Ook op wijd verspreide minicomputers en zelfs op vele persoonlijke computers is LISP nu beschikbaar. Dit boek kan dus nu gemakkelijker worden aangevuld met computerpraktijk.

Tenslotte is er nu ook een LISP standaard, m.n. COMMON LISP. Steeds meer implementaties convergeren naar deze standaard wat de portabiliteit van LISP programma's alleen maar vergroot.

Het boek is grondig herwerkt om aan de nieuwe standaard te voldoen. De wijzigingen liggen ook op het terminologische vlak. Bijvoorbeeld de notie van 'atoom' vervalt en wordt vervangen door die van 'symbool'. Fouten zijn verbeterd en de oefeningen zijn hier en daar uitgebreid. Functies en dan vooral de problematiek van lexical closures, krijgen meer aandacht. Ook is er nu een inleiding in LISP-omgevingen en de constructie van ingebedde talen. Nochtans wordt niet afgeweken van het oorspronkelijk opzet: een korte didactische inleiding in LISP voor informatici.

Suggesties voor verbetering zijn onder meer gekomen van Koenraad de Smedt en diverse wetenschappelijke medewerkers van de VUB. Nogmaals dank ook aan de uitgeversploeg van Academic Service die dit boek op een vlotte manier heeft geproduceerd en verdeeld.

Luc Steels
Brussel, 1 mei 1986

OVERZICHT

DEEL 1. BASISREPERTORIUM

1. SYMBOLEN EN LIJSTEN
2. FUNCTIES, FORMULES EN EVALUATIE
3. CAR EN CDR
4. CONS EN LIST
5. PREDIKATEN
6. COMBINATIES VAN PREDIKATEN
7. VOORWAARDELIJKE UITDRUKKINGEN
8. QUOTE, BACKQUOTE EN EVAL
9. READ EN PRINT
10. SAMENVATTING

Dit deel introduceert een basisrepertorium van elementaire LISP-bouwstenen. We beginnen met de primitieve datastructuren van LISP, symbolen en lijsten. Dan onderzoeken we hoe bekende wiskundige noties zoals functie, variabele, predikaat en voorwaardelijke uitdrukking worden uitgedrukt in LISP. Ook bekijken we enkele functies over de primitieve datastructuren. De oefeningen in dit deel zijn vooral bedoeld om te wennen aan de notatie.

1. SYMBOLEN EN LIJSTEN

1.1. DEFINITIES

LISP heeft diverse datatypen. Er zijn de datatypen die men standaard in de meeste programmeertalen aantreft, zoals getallen, strings, vectors, etc... We zullen deze typen later in detail behandelen. Getallen, zoals 12 of 3.1415, worden wel al gebruikt in de voorbeelden. LISP heeft daarnaast twee primitieve datatypen die van groot belang zijn voor het schrijven van symbolische programma's: het symbool en de lijst.

Een *symbool* wordt aangeduid met een reeks letters. Voorbeelden zijn ABC, abc, DIT-IS-EEN-SYMBOOL, -&. De letters mogen wel getallen bevatten, zoals in symbool-1, maar het mogen niet allemaal getallen zijn want dan krijgen we een object van het datatype getal.

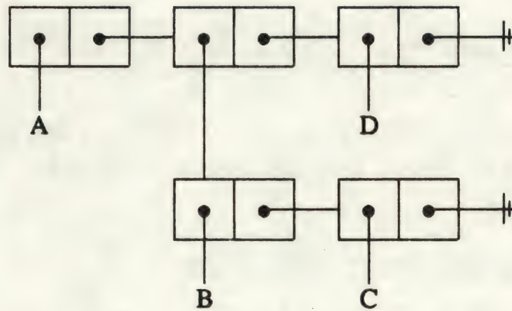
Het onderscheid tussen hoofd- en kleine letters speelt geen rol in de naam van een symbool. ABC is volledig equivalent aan abc of aBC. In de tekst worden gewoonlijk hoofdletters gebruikt voor LISP uitdrukkingen.

Een symbool is niet hetzelfde als een string omdat het een interne structuur heeft waaronder een eigenschapslijst. We zullen dit later nog bestuderen.

Een *lijst* is een reeks LISP-objecten gegroepeerd tussen haakjes. LISP-objecten zijn symbolen, objecten van andere datatypen, of opnieuw lijsten. (A B C), (12 (13 14) 15) zijn voorbeelden van lijsten.

Een lijst kan ook geen enkel element bevatten. Dit is de lege lijst of *nullijst*, voorgesteld als () of NIL. De nullijst wordt soms beschouwd als een symbool, soms als een lijst.

Lijsten zijn intern gerepresenteerd met cellen en pointers. Bijvoorbeeld de lijst (A (B C) D) is intern als volgt opgeslagen:



De LISP-gebruiker trekt zich hier echter niets van aan. De omzetting naar geheugencellen gebeurt volledig door het LISP systeem zelf en de gebruiker denkt enkel in termen van de symbolische notatie. Dit is overigens één van de grote voordelen van LISP. Door het automatisch geheugenbeheer wordt de programmeur bevrijd van deze moeilijke taak en kan hij zijn aandacht op andere aspecten van het programma vestigen.

1.2. OEFENINGEN

1. Is SYMBOOL een symbool?

Ja, want SYMBOOL is een reeks letters.

2. Is 450 een symbool?

Neen, 450 is een getal en dus geen symbool.

3. Is L een symbool?

Ja, want L is een reeks met 1 letter.

4. Is (SYMBOOL) een lijst?

Ja, want het is een symbool tussen haakjes.

5. Is (SYMBOOL EEN TWEE) een lijst?

Ja, want SYMBOOL, EEN en TWEE zijn symbolen en ze zijn gegroepeerd

tussen haakjes.

6. Is (SYMBOOL EEN TWEE) DRIE een lijst?

Nee, want alhoewel (SYMBOOL EEN TWEE) een lijst vormt en DRIE een symbool is, is het geheel geen lijst omdat de elementen niet tussen haakjes staan.

7. Is ((SYMBOOL EEN TWEE) DRIE) een lijst?

Ja, want (SYMBOOL EEN TWEE) is een lijst en DRIE is een symbool en het geheel staat tussen haakjes.

8. Hoeveel elementen bevat de lijst (EEN LIJST MET ELEMENTEN)?

Vier: EEN, LIJST, MET en ELEMENTEN.

9. Is het volgende een lijst: (((EEN) LIJST) MET ((ELEMENTEN)))?

Ja, want het is een reeks LISP-objecten tussen haakjes.

10. Is () een lijst?

Ja, het is een lijst zonder elementen, de nullijst.

11. Is (() () () ()) een lijst?

Ja, het is een reeks nulllijsten tussen haakjes.

12. Is het symbool SYMBOOL-1 gelijk aan het symbool symbool-1?

Ja, want hoofd- of kleine letters speelt geen rol.

2. FUNCTIES, FORMULES EN EVALUATIE

2.1. DEFINITIES

Het begrip van een functie is bekend uit de wiskunde. Een *functie* is een afbeelding die aan elk element of reeks elementen in het domein van de functie een unieke corresponderende waarde toekent. +, -, * (produkt), / (deling) zijn

voorbeelden van functies.

We hebben een manier nodig om op te schrijven dat we een functie willen toepassen op een reeks argumenten. Dit gebeurt in LISP via een lijst. Zo'n lijst is een eerste voorbeeld van een *formule*. Het eerste element is de naam van de functie. De overige elementen zijn de argumenten. Bijvoorbeeld (+ 1 2) is een lijst die aanduidt dat de functie + moet worden toegepast op de argumenten 1 en 2. Deze notatie heet prefix-notatie omdat de functie vooraan staat.

We hebben ook een manier nodig om *variabelen* te representeren. Dit gebeurt in LISP via een symbool. Een symbool dat functioneert als variabele is een tweede voorbeeld van een formule. Bijvoorbeeld L is een variabele in de uitdrukking (+ L 5). We zullen later zien hoe een waarde kan worden toegekend aan een variabele. Als een symbool een waarde heeft, zeggen we dat het *gebonden* is aan die waarde.

Merk op dat een variabele in LISP geen type heeft. Data-objecten hebben wel een type. "Type-checking" gebeurt op het moment dat het programma draait. Bijvoorbeeld de +-operator zal testen of de data-objecten waarop + wordt toegepast wel degelijk getallen zijn.

Een laatste belangrijke klasse van formules zijn speciale LISP-objecten zoals getallen of het symbool NIL die zichzelf voorstellen. Zij functioneren als constanten.

Een *formule* is dus ofwel een symbool dat een variabele voorstelt, een constante, ofwel een lijst met als eerste element de naam van een functie. De waarde van een formule wordt verkregen door haar te *evalueren*. Het evalueren van formules is *de* basisoperatie in LISP en we zullen dit proces nog grondig bestuderen in de volgende delen. Hier is een eerste definitie van evaluatie:

1. Als de formule een getal is of de nullijst, is de evaluatie van de formule gelijk aan de formule zelf. Bijvoorbeeld als de formule gelijk is aan 2, dan is de evaluatie van de formule gelijk aan 2.

2. Als de formule een symbool maar geen getal is, is de evaluatie van de formule gelijk aan de waarde van het symbool. Als het symbool geen waarde heeft, is de evaluatie gelijk aan UNDEFINED. Bijvoorbeeld als L gebonden is aan (A B), dan is de evaluatie van L gelijk aan (A B).

3. Als de formule een lijst is, is de evaluatie van de formule gelijk aan het toepassen van de functie op het resultaat van de evaluatie van de argumenten. Als de functie niet kan worden toegepast op de argumenten, is de evaluatie

van de formule gelijk aan UNDEFINED. Bijvoorbeeld, de evaluatie van (+ 1 2) is 3 omdat de evaluatie van 1 1 is, de evaluatie van 2 is 2, en + toegepast op 1 en 2 is gelijk aan 3.

Omdat de argumenten van een functie zelf ook formules zijn, laat dit evaluatieproces meteen toe composities van formules te evalueren. Bijvoorbeeld om (+ (+ 1 2) 4) te evalueren moeten we eerst (+ 1 2) en 4 evalueren. (+ 1 2) levert 3 op en 4 levert 4 op zodat de evaluatie van (+ (+ 1 2) 4) gelijk is aan 7.

Als evaluatie gelijk is aan UNDEFINED, wordt een fouttoestand gegenereerd door het LISP systeem. We zullen dit later uitvoeriger behandelen.

Eén van de belangrijkste eigenschappen van deze notatie is dat formules LISP-datastructuren zijn. Het is dus mogelijk dat een LISP formule een ander formule construeert of modificeert en dat deze formule daarna wordt geëvalueerd.

2.2. OEFENINGEN

1. Wat is de evaluatie van (- 10 5)?

5, omdat de evaluatie van 10 10 is, de evaluatie van 5 is 5 en - toegepast op 10 en 5 is 5.

2. Wat is de evaluatie van (- 5 10)?

-5, omdat - toegepast op 5 en 10 gelijk is aan -5.

3. Wat is de evaluatie van (+ (- 9 5) 3)?

7, omdat de evaluatie van (- 9 5) gelijk is aan 4, de evaluatie van 3 is 3 en + toegepast op 4 en 3 is gelijk aan 7.

4. Wat is de evaluatie van (* (/ 10 2) 2)?

10, want de evaluatie van (/ 10 2), d.w.z. 10 gedeeld door 2, is 5 en * toegepast op 5 en 2 is 10.

5. Gegeven L gebonden aan 5 en M gebonden aan 3, wat is de evaluatie van (+ L M)?

8, omdat de evaluatie van L 5 is, de evaluatie van M 3 is, en + toegepast op

5 en 3 gelijk is aan 8.

6. Schrijf een formule die - toepast op 16 en 3.

(- 16 3).

7. Gegeven L gebonden aan 4 wat is de evaluatie van (- K L)?

UNDEFINED want de evaluatie van K is UNDEFINED.

8. Wat is L als de evaluatie van (+ L 5) 7 is?

2.

3. CAR en CDR

3.1. DEFINITIES

Er zijn twee primitieve functies om delen van een lijst te adresseren: CAR en CDR

1. De CAR van een lijst is gelijk aan het eerste element van de lijst. De CAR van de nullijst is de nullijst. De CAR van een symbool is ongedefinieerd.
2. De CDR van een lijst is gelijk aan de lijst behalve het eerste element. De CDR van de nullijst is de nullijst. De CDR van een symbool is ongedefinieerd.

De functienamen komen van de eerste IBM 704 implementatie van LISP. CAR is een afkorting van Contents of Address Register en CDR is een afkorting van Contents of Decrement Register. CDR wordt als 'kudder' uitgesproken, of soms 'koeder'.

Achtereenvolgende toepassingen van CAR of CDR worden afgekort door A of D te schrijven tussen C · · · R. Bijvoorbeeld (CAR (CAR (CDR X))) is gelijk aan (CAADR X). Gewoonlijk is het afkorten beperkt tot vier composities. Bijvoorbeeld (CAAAAAAR X) is ongedefinieerd omdat er zes composities zijn.

3.2. OEFENINGEN

1. Gegeven L gebonden aan (((A)) (B) (C) D), wat is de evaluatie van (CAR L)?

((A)) want ((A)) is het eerste element van de lijst L.

2. Gegeven L gebonden aan (((A)) (B)), wat is de evaluatie van (CAR (CAR L))?

(A), want het eerste element van L is ((A)) en het eerste element hiervan is (A).

3. Gegeven L gebonden aan (A B C) wat is (CDR L)?

(B C) want de elementen in de lijst zonder A zijn (B C).

4. Gegeven L gebonden aan ((A B C) X Y Z) wat is (CDR L)?

(X Y Z) want het eerste element van L is (A B C).

5. Gegeven A gebonden aan SYMBOOL, wat is (CAR A)?

UNDEFINED, want de CAR van een symbool is ongedefinieerd.

6. Gegeven A gebonden aan SYMBOOL, wat is (CDR A)?

UNDEFINED, want de CDR van een symbool is ongedefinieerd.

7. Gegeven L gebonden aan (), wat is (CAR L)?

(), het eerste element van de nullijst is de nullijst.

8. Gegeven L gebonden aan ((B) (A C) ((D))), wat is (CAR (CDR L))?

(A C) want (CDR L) is gelijk aan ((A C) ((D))) en de car hiervan is (A C).

9. Gegeven L gebonden aan ((B) (A C) ((D))), wat is (CDR (CDR L))?

(((D))) want (CDR L) is gelijk aan ((A C) ((D))) en de CDR hiervan is (((D))).

10. Gegeven L gebonden aan (A (B (C)) D), wat is (CDR (CAR L))?

UNDEFINED, want (CAR L) is gelijk aan A en de CDR van een symbool is

ongedefinieerd.

11. Waarvan is (CAADAR X) de afkorting?

(CAR (CAR (CDR (CAR X))))

12. Wat is de afkorting van (CAR (CDR (CDR X)))?

(CADDR X).

13. Gegeven een lijst L gebonden aan ((A B) C) welke formule levert B op?

(CADAR L), want de CAR van L is gebonden aan (A B), de CDR hiervan is gelijk aan (B) en de CAR hiervan is B.

14. Gegeven een lijst L gebonden aan ((A) (B (C) D)) welke formule levert C op?

(CAADR (CADR L)) want de CDR van L is ((B (C) D)), de CAR hiervan is (B (C) D), de CDR hiervan is ((C) D), de CAR hiervan is (C) en de CAR hiervan is C.

4. CONS EN LIST

4.1. DEFINITIE

De primitieve functie om nieuwe lijsten te maken is CONS (denk aan *cons*trueer). CONS heeft twee argumenten. Het voegt het eerste argument toe aan het tweede argument. Het tweede argument van CONS is een lijst. Als het tweede argument van CONS een symbool is, is het resultaat van de evalutie gelijk aan UNDEFINED[1]. Bijvoorbeeld, gegeven L gebonden aan (A B) en M gebonden aan C, dan is de evaluatie van (CONS M L) gelijk aan (C A B).

Er is ook een meer algemene functie LIST die een lijst maakt van de evaluatie van zijn argumenten. Bijvoorbeeld

```
(LIST 5 4 2 (+ 1 5))
```

---

[1] Normaal is in dit geval het resultaat gelijk aan een "dotted pair". Bijvoorbeeld (CONS M L) met M gebonden aan A en L gebonden aan B is gelijk aan (A . B). Om pedagogische redenen worden dotted pairs in dit boek niet behandeld.

is gelijk aan (5 4 2 6). Als LIST geen argumenten heeft is het resultaat de lege lijst.

4.2. OEFENINGEN

1. Gegeven A gebonden aan SYMBOOL en L gebonden aan (EEN LIJST) wat is (CONS A L)?

   (SYMBOOL EEN LIJST), want SYMBOOL wordt toegevoegd aan de lijst (EEN LIJST).

2. Gegeven L gebonden aan (EEN LIJST) en M gebonden aan (EN NOG EEN LIJST), wat is (CONS L M)?

   ((EEN LIJST) EN NOG EEN LIJST).

3. Gegeven L gebonden aan ((A) B) en M gebonden aan (C D ((E) F G)), wat is (CONS L M)?

   (((A) B) C D ((E) F G))

4. Wat zijn de argumenten van CONS?

   CONS heeft twee argumenten: een LISP-object en een lijst.

5. Gegeven L gebonden aan (A B (C)) en M gebonden aan (), wat is (CONS L M)?

   ((A B (C))).

6. Gegeven L gebonden aan A en M gebonden aan () wat is (CONS L M)?

   (A)

7. Gegeven A gebonden aan EEN en L gebonden aan ((SYMBOOL) LIJST), wat is (CONS A (CAR L))?

   (EEN SYMBOOL) omdat (CAR L) gebonden is aan (SYMBOOL) en EEN

hieraan toegevoegd (EEN SYMBOOL) oplevert.

8. Gegeven A gebonden aan EEN en L gebonden aan ((SYMBOOL) LIJST), wat is (CONS A (CDR L))?

(EEN LIJST), want (CDR L) is (LIJST) en EEN hieraan toegevoegd levert (EEN LIJST) op.

9. Gegeven L gebonden aan EEN en M gebonden aan LIJST. Welke formule levert (EEN LIJST) op?

(CONS L (CONS M NIL)), want (CONS M NIL) levert (LIJST) op en L hieraan toegevoegd is gelijk aan (EEN LIJST).

10. Gegeven L gebonden aan (EEN LIJST) welke formule levert (LIJST EEN) op?

(CONS (CADR L) (CONS (CAR L) NIL)), want (CADR L) is gelijk aan LIJST en (CONS (CAR L) NIL) is gelijk aan (EEN). (LIST (CADR L) (CAR L)) bereikt hetzelfde resultaat.

11. Wat is (LIST (+ 5 4) 1 (+ 9 10))?

(9 1 19).

5. PREDIKATEN

5.1. DEFINITIES

Een predikaat is een functie die een bepaalde conditie test en NIL oplevert als die conditie niet waar is. Als die wel waar is, levert evaluatie een waarde op die niet gelijk is aan NIL (maar het is verder niet bepaald wat). Dikwijls wordt het bijzondere symbool T gebruikt voor waar. T is een afkorting van 'True' (Engels voor 'waar') en is een constante. Het is gebruikelijk de naam van een predikaat te laten eindigen op P.

Hier zijn enkele voorbeelden van predikaten:

1. De evaluatie van (NULL S) is gelijk aan T als S aan de nullijst gebonden is, anders NIL.

2. De evaluatie van (SYMBOLP S) is gelijk aan T als S een symbool is, anders NIL.

3. De evaluatie van (EQ L M) is gelijk aan T als L gelijk is aan M, anders NIL.

4. De evaluatie van (NUMBERP S) is gelijk aan T als S een getal is, anders NIL.

5. < and > leveren T op als de argumenten kleiner, respectievelijk groter zijn, anders NIL. Als de argumenten geen getallen zijn krijgen we UNDEFINED.

6. = test gelijkheid tussen getallen. (= X Y) is waar als X en Y numerisch gelijk zijn.

5.2. OEFENINGEN

1. Gegeven dat L gebonden is aan NIL, is (NULL L) gelijk aan T?

   Ja, want NIL is de nullijst.

2. Gegeven dat L gebonden is aan (EEN LIJST), wat is (NULL L) ?

   NIL, want L is niet de nullijst.

3. Gegeven dat L gebonden is aan SYMBOOL, wat is (NULL L)?

   NIL, want L is niet de nullijst.

4. Gegeven A gebonden aan SYMBOOL, is de evaluatie van A een symbool, m.a.w. is (SYMBOLP A) T?

   Ja, want SYMBOOL is een symbool.

5. Gegeven A gebonden aan (EEN LIJST), wat is (SYMBOLP A)?

   NIL, want (EEN LIJST) is geen symbool.

6. Gegeven L gebonden aan (EEN LIJST SYMBOLEN), wat is (SYMBOLP (CAR L))?

T, want (CAR L) is gelijk aan EEN en EEN is een symbool.

7. Gegeven A gebonden aan JORIS en B gebonden aan JORIS, wat is (EQ A B)?

T, want JORIS is gelijk aan JORIS.

8. Gegeven A gebonden aan JORIS en B gebonden aan KAREL, wat is (EQ A B)?

NIL, want JORIS is niet gelijk aan KAREL.

9. Gegeven A gebonden aan JORIS en B gebonden aan (KAREL PIET), wat is (EQ A B)?

NIL.

10. Gegeven L gelijk aan (EEN EEN LIJST LIJST), wat is (EQ (CAR L) (CAR (CDR L)))?

T, want (CAR L) is EEN, (CDR L) is (EEN LIJST LIJST) en dus (CAR (CDR L)) is EEN. EEN is gelijk aan EEN.

11. Wat is (NUMBERP 15)?

T, want 15 is een getal.

12. Gegeven A gebonden aan SYMBOOL, wat is (NUMBERP A)?

NIL, want SYMBOOL is geen getal.

13. Wat is (< 15 6)?

NIL want 15 is niet kleiner dan 6.

14. Wat is (> 15 6)?

T want 15 is groter dan 6.

15. Wat is (= X Y) met X gebonden aan 5 en Y gebonden aan 6.

NIL want X is niet gelijk aan Y.

6. COMBINATIES VAN PREDIKATEN

6.1. DEFINITIES

Toepassingen van predikaten kunnen gecombineerd worden met de logische connectieven AND, OR and NOT. Deze connectieven zijn zelf ook weer functies, zogenaamde propositionele functies, en ze worden op dezelfde manier opgeschreven als alle andere functies. De connectieven zijn als volgt gedefinieerd:

1. De evaluatie van (NOT M) is gelijk aan T als M gelijk is aan NIL anders NIL. Bijvoorbeeld (NOT 2) is NIL.
2. De evaluatie van (AND $M_1 \cdots M_n$) is gelijk aan de evaluatie van M_n als geen enkel van de argumenten evalueert naar NIL, anders is het gelijk aan NIL. Bijvoorbeeld (AND NIL T) is gelijk aan NIL.
3. De evaluatie van (OR $M_1 \cdots M_n$) is gelijk aan de evaluatie van de eerste formule M die *niet* gelijk is aan NIL, anders is het gelijk aan NIL. Bijvoorbeeld (OR NIL T) is gelijk aan T.

6.2. OEFENINGEN

1. Wat is de evaluatie van (AND NIL NIL)?

NIL.

2. Wat is de evaluatie van (OR T NIL NIL)?

T, want T is het eerste argument dat niet gelijk is aan NIL.

3. Wat is de evaluatie van (NOT T)?

NIL, want het argument van NOT is niet gelijk aan NIL.

4. Wat is (AND (OR T (NOT T)) NIL)?

NIL want dit is het eerste argument van AND dat niet gelijk is aan T.

5. Gegeven L gebonden aan NIL en M gebonden aan T, construeer een formule waarin L en M voorkomen en waarvan de evaluatie gelijk is aan T.

(OR L M) of (AND (NOT L) M).

6. Wat is (NOT (NUMBERP NIL))?

T, want (NUMBERP NIL) is NIL and (NOT NIL) is T.

7. Welke bindingen kan L hebben om (OR (NULL L) (SYMBOLP L)) gelijk te maken aan T?

L kan NIL zijn of gelijk welk symbool.

8. Wat is
```
(AND (NUMBERP 15)
     (NOT
       (EQ
         (CAR L)
         (CADR L))))
```
met L gebonden aan (A B C)?

T want (NUMBERP 15) is T, (EQ (CAR L) (CADR L)) is NIL omdat A verschillend is van B, en (NOT (EQ (CAR L) (CADR L))) is gelijk aan T.

7. VOORWAARDELIJKE UITDRUKKINGEN

7.1. DEFINITIES

Een voorwaardelijke uitdrukking is een uitdrukking die een resultaat oplevert afhankelijk van een conditie. Wiskundigen schrijven dit op met een accolade. Bijvoorbeeld: de definitie van de functie ABS voor de absolute waarde van een getal x is:

$$\text{abs}(x) = \begin{cases} x & \text{als } x \geq 0 \\ -x & \text{als } x < 0 \end{cases}$$

$X \geq 0$ en $X < 0$ zijn condities. X of -X is het resultaat als 1 van de condities waar is.

De functie COND dient om een voorwaardelijke uitdrukking te noteren in LISP. COND heeft een onbepaald aantal argumenten. Elk argument is een lijst met een predikaat P_i dat een conditie uitdrukt en een lijst van formules $Q_{i,j}$:

(COND
 $(P_1 \;\; Q_{1,1} \; Q_{1,2} \cdots Q_{1,m_1})$
 $(P_2 \;\; Q_{2,1} \; Q_{2,2} \cdots Q_{2,m_2})$
 . . .
 $(P_n \;\; Q_{n,1} \; Q_{n,2} \cdots Q_{n,m_n}))$

De predikaten van elk argument worden na elkaar geëvalueerd, tot er één niet NIL oplevert. Als een predikaat P_i niet NIL oplevert, worden de formules $Q_{i,1} \ldots Q_{i,m_i}$ geëvalueerd. Het resultaat van het geheel is gelijk aan het resultaat van de laatst geëvalueerde formule. Als geen enkel predikaat T oplevert, is het geheel NIL. Bijvoorbeeld als het predikaat P_3 waar oplevert, is het resultaat van de voorwaardelijke uitdrukking gelijk aan de evaluatie van Q_{3,m_3}. Een conditie gelijk aan T wil zoveel zeggen als "anders", omdat de corresponderende formule altijd wordt geëvalueerd. Daarom komt T dikwijls voor als laatste conditie.

De functie ABS ziet er bijvoorbeeld als volgt uit in LISP:

```
(COND
  ((< X 0) (- X))
  (T X))
```

Als X kleiner is dan 0, dit wil zeggen als x negatief is, is het resultaat -X, anders X.

Een veel gebruikte syntactische variant van een conditionele uitdrukking is

```
(IF <conditie> <formule-als-waar> <formule-als-onwaar>)
```

Het resultaat van evaluatie van deze formule is gelijk aan het resultaat van de evaluatie van <formule-als-waar> als de evaluatie van <conditie> non-NIL is.

De functie ABS ziet er bijvoorbeeld als volgt uit:

```
(IF (< X 0) (- X) X)
```

7.2. OEFENINGEN

1. (COND
   ((NULL L) T)
   ((SYMBOLP L) L)
   (T (CAR L)))

   is een voorbeeld van een voorwaardelijke uitdrukking. Gegeven dat L gebonden is aan NIL, wat is het resultaat?

   T, want NIL is gelijk aan de nullijst en daarom is (NULL L) waar.

2. Gegeven dezelfde voorwaardelijk uitdrukking, wat is het resultaat als L gebonden is aan SYMBOOL?

   SYMBOOL, want (NULL L) levert NIL op en (SYMBOLP L) levert T op. Het resultaat is gelijk aan de evaluatie van L en dus SYMBOOL.

3. Gegeven dezelfde voorwaardelijke uitdrukking, wat is het resultaat als L gebonden is aan (A B C).

   A, want (NULL L) is NIL, (SYMBOLP L) is ook NIL en T is altijd waar, dus is het resultaat gelijk aan (CAR L) en dus A.

4. Beschrijf de voorwaardelijke uitdrukking in 1.

   De uitdrukking levert T op als L een nullijst is, L zelf als L een symbool is anders het eerste element van L.

5. Construeer een voorwaardelijke uitdrukking die gegeven twee getallen A en B, de som van A en B oplevert als de getallen gelijk zijn aan elkaar, anders het verschil.

```
(COND
  ((= A B) (+ A B))
  (T (- A B)))
```

6. Zet deze formule om in een formule die IF gebruikt.

```
(IF (EQ A B) (+ A B) (- A B))
```

7. Construeer een voorwaardelijke uitdrukking die A toevoegt aan L als het eerste element van L gebonden is aan A.

Analyse: Als L de lege lijst is zal het eerste element zeker niet gelijk zijn aan A. Als het eerste element gelijk is aan A voegen we A toe, anders moet de voorwaardelijke uitdrukking L zelf opleveren.

```
(COND
  ((NULL L) L)
  ((EQ (CAR L) A) (CONS A L))
  (T L))
```

8. Construeer een voorwaardelijke uitdrukking die nagaat of het eerste element van een lijst L gelijk is aan het tweede element.

Analyse: Als de lijst L de lege lijst is of slechts 1 element bevat kan het eerste element niet gelijk zijn aan het tweede element want er zijn onvoldoende elementen. Anders vergelijken we het eerste element met het tweede element.

```
(COND
  ((OR
    (NULL L)
    (NULL (CDR L)))
   NIL)
  (T
    (EQ
      (CAR L)
      (CAR (CDR L)))))
```

9. Schrijf een voorwaardelijke uitdrukking die de code van een LISP-object L geeft. De code is 0 als L gebonden is aan een getal, 1 aan een symbool maar geen getal en 2 aan een lijst.

```
(COND
  ((NUMBERP L) 0)
  ((SYMBOLP L) 1)
  (T 2))
```

8. QUOTE, BACKQUOTE EN EVAL

8.1. DEFINITIES

Soms is het nodig om de evaluatie van een LISP-object te vermijden. Bijvoorbeeld, als we CAR willen toepassen op (A B) was het tot hiertoe noodzakelijk om een variabele, bijv. L, te veronderstellen, die gebonden is aan (A B). Het zou eenvoudiger zijn als we CAR direct op (A B) kunnen toepassen zonder eerst een variabele te introduceren. Dat kan echter niet door te zeggen (CAR (A B)) want dan zou (A B) geëvalueerd worden en A is geen functie.

De functie QUOTE verhindert evaluatie. Bijvoorbeeld in plaats van (CAR L) kunnen we zeggen (CAR (QUOTE (A B))). Evaluatie van (CAR (QUOTE (A B))) begint met de evaluatie van het argument (QUOTE (A B)). Evaluatie van (QUOTE (A B)) is gelijk aan (A B) en dus de evaluatie van (CAR (QUOTE (A B))) is gelijk aan A.

De afkorting van (QUOTE ...) is '... Bijvoorbeeld (CAR (QUOTE (A B))) is gelijk aan (CAR '(A B)).

Naast QUOTE is er ook een inverse functie die evaluatie expliciet oproept. De functie EVAL heeft 1 argument, een LISP-object, en levert de evaluatie van deze formule als waarde. Dus

```
(EVAL (+ 5 10))
```

is gelijk aan 15. Merk op dat de argumenten van EVAL al een eerste maal geëvalueerd zijn alvorens EVAL ze evalueert, net zoals het argument van CAR al geëvalueerd wordt alvorens CAR zelf in actie treedt. Dus in feite krijgt EVAL het getal 15 binnen. De evaluatie van 15 is 15, dus is het eindresultaat 15.

```
(EVAL '(+ 5 10))
```

is ook gelijk aan 15 omdat de evaluatie van (+ 5 10) 15 is.

Het gebruik van quote en eval komt veel voor bij het maken van lijsten. Vandaar dat een syntactische variant is gegroeid die het gebruik van deze functies combineert. Dit is de hulpfunctie ‘, BACKQUOTE, die hetzelfde doet als quote,

d.w.z. '(A B C) is gelijk aan (A B C), behalve dat LISP-objecten voorafgegegaan door een komma *wel* geëvalueerd worden. Dus '(A ,(+ 2 3) (+ 2 3)) is gelijk aan (A 5 (+ 2 3)). Als het element na de komma een lijst is kunnen de elementen één voor één toegevoegd worden door ,@ te gebruiken in de plaats van ,. Dus '(A ,@'(A B C)) is gelijk aan (A A B C).

8.2. OEFENINGEN

1. Wat is de evaluatie van (CONS 'A '(B C))?

   (A B C), want de evaluatie van 'A is A, de evaluatie van '(B C) is (B C) en CONS toegepast op A en (B C) levert (A B C) op.

2. Wat is de formule die CONS toepast op de lijsten (C D) en (D E)?

   (CONS '(C D) '(D E)).

3. Wat is (EQ 'A (CAR (CDR '(B A C))))?

   T, want (CDR '(B A C)) is (A C). CAR hierop toegepast is A en A is gelijk aan A.

4. Schrijf een voorwaardelijke uitdrukking die L oplevert als L gebonden is aan A, anders NIL.

   (IF (EQ L 'A) L NIL)

5. Wat is (CONS A '(B C))?

   UNDEFINED want A is niet gebonden.

6. Wat is de evaluatie van (CAR (A B))?

   UNDEFINED, want de evaluatie van (A B) is gelijk aan het toepassen van de functie A op de evaluatie van B. De evaluatie van B is UNDEFINED want B is niet gebonden en bovendien is de functie A ongedefinieerd.

7. Moet er een '-teken voor T of NIL in (AND T NIL)?

Nee, want de evaluatie van T levert T op en de evaluatie van NIL is gelijk aan NIL.

8. Gegeven L gebonden aan (A B), wat is de evaluatie van (CONS 'L NIL)?

(L), want de evaluatie van 'L is L en L toegevoegd aan de lege lijst geeft (L). Merk op dat (CONS 'L NIL) niet gelijk is aan ((A B))!

9. Wat is de evaluatie van (EVAL '(CONS 'A NIL))?

(A) want EVAL moet (CONS 'A NIL) evalueren en dit is gelijk aan (A).

10. Wat is de evaluatie van (EVAL (LIST 'CAR (CDR '(A))))?

NIL, want het argument van EVAL, namelijk (LIST 'CAR (CDR '(A))), is gelijk aan (CAR NIL). De evaluatie hiervan is gelijk aan NIL.

11. Wat is de evaluatie van (EVAL (CONS '+ (CONS 1 '(2))))

3, want het argument van EVAL is (+ 1 2) en de evaluatie hiervan is gelijk aan 3.

12. Wat is de evaluatie van (EVAL (CAR '(A)))

UNDEFINED, want de evaluatie van (CAR '(A)) is A, en de evaluatie van A is ongedefinieerd omdat A niet gebonden is.

13. Wat is '(A B C)?

(A B C).

14. Wat is '(A B ,(+ 1 2))?

(A B 3).

15. Wat is '(,@(CONS 'A NIL) A ,(CONS 'A NIL))?

(A A (A)) De eerste A komt van de evaluatie van (CONS 'A NIL) dat (A) oplevert. Omdat er een @-teken voor de uitdrukking staat moeten de

elementen echter apart toegevoegd worden aan het eind-resultaat.

16. Gegeven L gelijk aan '(A B C), construeer een uitdrukking die ((A B C) A B C) oplevert via een backquote.

'(,L ,@L).

9. READ en PRINT

9.1. DEFINITIES

Een LISP-systeem bestaat normaal uit een zich hernemende READ-EVAL-PRINT cyclus. Dit wil zeggen dat het systeem een uitdrukking leest, die uitdrukking evalueert volgens de evaluatieregels, en het resultaat dan uitdrukt. Daarna begint de cyclus opnieuw. Een LISP-systeem gedraagt zich dus als een rekenmachine waar we dingen intikken en het resultaat direct kunnen zien.

Dit werkt erg goed voor de ontwikkeling van programma's omdat we functies individueel kunnen uittesten, eventueel wijzigen, en direct opnieuw testen zonder door een compile-link-load-execute cyclus te moeten gaan. Als we toch uitgeteste functies snel willen doen werken kunnen we onderdelen compileren - d.w.z. omzetten in machinetaal en klaar maken voor snelle uitvoering - en ze gebruiken alsof ze behoren tot de primitieve componenten van LISP.

READ en PRINT zijn functies zoals CAR en CONS. De functie READ, zonder argumenten, leest een LISP-object van de console die de gebruiker verbindt met het LISP-systeem. READ kan een extra argument hebben dat een ander medium (bijvoorbeeld een file op een schijf) aanduidt. De conventies voor het beschrijven van dit medium hangen af van de LISP-implementatie en het uitbatingssysteem.

READ blijft lezen tot een volledige LISP-object is bereikt zelfs als die meer dan één lijn in beslag neemt. Extra spaties spelen geen rol. Er zijn een aantal bijzondere tekens waar READ rekening mee houdt. Voorlopig bespreken we ; en \.

READ negeert alles na ;. Dit is dus een manier om commentaar toe te voegen aan een programma. Bijvoorbeeld

```
(COND
  ((= A B)            ; als A gelijk is aan B
   (+ A B))           ; tel A op bij B
  (T (- A B)))        ; anders bereken het verschil van A en B
```

is een formule met commentaar.

READ neemt het karakter dat na een \ komt als teken in plaats van symbool. Bijvoorbeeld, A\(B is een geldig symbool gelijk aan A(B. Het linkerhaakje wordt beschouwd als het teken '(' en niet als "linkerhaakje". Dit is ook de manier om het onderscheid tussen hoofd en kleine letters expliciet te maken in de naam van een symbool. A\bC is dus niet hetzelfde als A\BC. In het eerste is de tweede letter van de naam een kleine letter, in het tweede een hoofdletter.

De functie PRINT heeft 1 argument en zorgt ervoor dat de evaluatie van dit argument op het scherm verschijnt. Bijvoorbeeld na het typen van

```
(PRINT (+ 5 10))
```

verschijnt 15 op het scherm. PRINT kan nog een tweede argument hebben dat zoals bij READ aangeeft waar de output naar toe moet. Het resultaat van (PRINT <object>) is altijd <object>.

Een stap in de READ-EVAL-PRINT cyclus van een LISP-systeem kan nu beschreven worden als

```
(PRINT (EVAL (READ)))
```

Deze formule leest een LISP-object, evalueert het, en drukt het resultaat uit.

9.2. OEFENINGEN

1. Schrijf een formule die twee getallen leest en ze bij elkaar optelt.

```
(+ (READ) (READ))
```

2. Wat gebeurt er tijdens de evaluatie van (AND (PRINT T) (PRINT 'A))?

Eerst verschijnt T op het scherm. Het resultaat van (PRINT T) is T, dus AND gaat het tweede argument evalueren. Dit heeft tot resultaat dat A op het scherm verschijnt. (PRINT 'A) levert ook T, zodat het eindresultaat gelijk is aan T.

3. Wat is de evaluatie van

```
(READ)
(DIT IS EEN LANG LISP-OBJECT              ; eerste deel
 DAT TWEE LIJNEN IN BESLAG NEEMT)         ; tweede deel
```

(DIT IS EEN LANG LISP-OBJECT DAT TWEE LIJNEN IN BESLAG

NEEMT). Het gedeelte na de ; is geen deel van het gelezen LISP-object.

4. Is (EQ Dit-hier \dit-hier) T?

Nee, want het eerste karakter van het tweede symbool is expliciet een kleine letter.

5. Is het mogelijk een symbool te maken met de naam (A B)?

Ja. \(\A \ \B\).

10. SAMENVATTING

Een symbool is een reeks letters of een getal. Een lijst is een reeks LISP-objecten gegroepeerd tussen haakjes. Een LISP-object is een symbool of een lijst.

+, -, * en / zijn functies voor som, verschil, vermenigvuldiging en deling.

De CAR van een lijst is gelijk aan het eerste element. De CDR van een lijst is gelijk aan de lijst behalve het eerste element. CONS voegt een element bij een lijst. LIST groepeert zijn argumenten in een lijst.

NULL is een predikaat dat waar oplevert als het argument de nullijst is. SYMBOLP is een predikaat dat waar oplevert als het argument een symbool is. EQ is een predikaat dat waar oplevert als het eerste argument gelijk is aan het tweede argument[2]. NUMBERP is een predikaat dat waar oplevert als het argument een getal is. <, >, en = testen of twee getallen kleiner, groter of gelijk zijn.

AND, OR en NOT dienen om predikaten te combineren.

COND ligt aan de basis van de voorwaardelijke uitdrukking. Een variant van COND is IF.

QUOTE verhindert evaluatie. EVAL zorgt voor evaluatie. BACKQUOTE combineert QUOTE en selectieve EVAL.

READ en PRINT lezen en schrijven een LISP-object.

---

[2] Alhoewel EQ ook kan gebruikt worden om lijsten te vergelijken, beperken we het gebruik ervan voorlopig tot die gevallen waar we de gelijkheid van een symbool met een LISP-object willen testen.

DEEL 2. FUNCTIEDEFINITIES

1. HET DEFINITIEMECHANISME
2. DYNAMISCH EN LEXICAAL BEREIK
3. SUBSTITUTIEFUNCTIES
4. RECURSIEVE DEFINITIES I: STAARTRECURSIE
5. RECURSIEVE DEFINITIES II: CONSTRUCTIEVE RECURSIE
6. RECURSIEVE DEFINITIES III: RECURSIE MET ACCUMULATOREN
7. VERSLAG
8. SAMENVATTING
9. GEMENGDE OPGAVEN

Echt programmeren begint als we het basisrepertorium van LISP-functies zelf uitbreiden met nieuwe functies. In dit deel bekijken we hoe dit kan. We bestuderen ook een aantal schema's om functies te definiëren, voornamelijk van het recursieve type.

1. HET DEFINITIEMECHANISME

1.1. DEFINITIES

DEFUN is een functie om een nieuwe functie te definiëren. DEFUN heeft drie componenten: de naam van de nieuwe functie, een lijst van de argumenten (ook soms formele variabelen genoemd) en de definitie zelf. Deze componenten worden als volgt genoteerd:

```
(DEFUN functie argumenten definitie)
```

De definitie bestaat uit minstens één formule.

Bijvoorbeeld de volgende uitdrukking:

```
(DEFUN ABS (X)
  (IF (< X 0) (- X) X))
```

definieert de functie ABS die de absolute waarde van een getal berekent. Na het evalueren van deze uitdrukking kunnen we ABS gebruiken alsof het een primitieve LISP-functie is.

Als de evaluator de opdracht krijgt om een functie toe te passen die met DEFUN is gedefinieerd, moet hij de argumenten van de definitie binden aan het resultaat van de evaluatie van de argumenten in het oproepen van de functie en dan de definitie zelf evalueren. Bijvoorbeeld voor

```
(ABS 5)
```

wordt X gebonden aan 5 en dan wordt de definitie van ABS, namelijk

```
(IF (< X 0) (- X) X)
```

geëvalueerd. Het resultaat is 5 omdat (< X 0) onwaar is.

Het is erg belangrijk om goed te begrijpen wat het bereik is van de variabelen in de opeenvolgende oproepen van functies door de evaluator. Bij het binnenkomen in de functie worden de variabelen van de definitie gebonden aan de waarden meegegeven tijdens de oproep. Bij het buitengaan uit de functie zijn deze bindingen niet langer meer geldig en worden de bindingen die geldig waren vóór de oproep terug van kracht. Bijvoorbeeld in

(ABS (ABS -5))

is X gebonden aan -5 in de tweede oproep, en aan 5 in de eerste oproep.

Er zijn twee redenen om een nieuwe functie te introduceren:

i. om te vermijden dat dezelfde combinatie van operaties meer dan eens moet worden beschreven, en

ii. om structuur aan te brengen in programma's.

Dit laatste punt is erg belangrijk. Door een afzonderlijke functie te definiëren vermindert de complexiteit van de formules waarin deze functie voorkomt.

1.2. OEFENINGEN

1. Welke formule levert het tweede element van een lijst L op?

(CAR (CDR L))

2. Definieer een functie TWEEDE die het tweede element van een lijst oplevert.

(DEFUN TWEEDE (L) (CAR (CDR L)))

3. Gegeven de formule (TWEEDE '(A B C)), wat is de eerste stap in de evaluatie?

L wordt gelijk gezet aan (A B C) en (CAR (CDR L)) wordt geëvalueerd.

4. Wat is het resultaat van de evaluatie van (TWEEDE '(A B C))?

B, want (CAR (CDR L)) met L gelijk aan (A B C) is B.

5. Schrijf een voorwaardelijke uitdrukking die T oplevert als het eerste element van een lijst L een getal is, anders NIL.

```
(COND ((NUMBERP (CAR L)) T)
      (T NIL))
```

6. Definieer nu een functie IS-EERSTE-ELEMENT-GETAL met deze voorwaardelijke uitdrukking.

```
(DEFUN IS-EERSTE-ELEMENT-GETAL (L)
  (COND ((NUMBERP (CAR L)) T)
        (T NIL)))
```

7. Gegeven de formule (IS-EERSTE-ELEMENT-GETAL '(A B C)), wat is de eerste stap in de evaluatie?

L wordt gelijk gezet aan (A B C) en de formule

```
(COND ((NUMBERP (CAR L)) T)
      (T NIL))
```

wordt geëvalueerd.

8. Wat is de volgende stap?

Het eerste predikaat in de voorwaardelijke uitdrukking, namelijk (NUMBERP (CAR L)), wordt geëvalueerd. Omdat (CAR L) gelijk is aan A levert dit NIL op.

9. Wat is de volgende stap?

Het tweede predikaat in de voorwaardelijke uitdrukking, namelijk T, wordt geëvalueerd. De evaluatie van T is T, zodanig dat het eindresultaat van

```
(COND ((NUMBERP (CAR L)) T)
      (T NIL))
```

gelijk is aan NIL.

2. DYNAMISCH EN LEXICAAL BEREIK

2.1. DEFINITIES

We hebben gezien dat er bindingen worden gemaakt tussen de formele variabelen en de oproepargumenten bij het binnenkomen van een functie en dat deze bindingen ongeldig worden bij het buitengaan. Er is verder een onderscheid tussen LISP-systemen wat betreft het bereik van de variabelen voor de evaluatie van

deeluitdrukkingen van een functie.

Een LISP-systeem heeft een *lexicaal* bereik als de variabelen enkel maar gebonden zijn *binnen* een functiedefinitie *als ze argumenten zijn van de functiedefinitie*. Bijvoorbeeld, gegeven

```
(DEFUN FOO (X Y) Z)
```

dan leidt de evaluatie van een formule met FOO altijd tot UNDEFINED omdat het evaluatiemechanisme Z niet bindt. Dit bereik heet lexicaal omdat alle bindingen binnen dezelfde tekst aanwezig moeten zijn. De term 'statisch' wordt soms gebruikt in plaats van lexicaal.

Sommige (vooral oudere) LISP-systemen hebben een *dynamisch* bereik. Dit wil zeggen dat variabelen die gebonden zijn tijdens het evaluatieproces gebonden blijven voor deeluitdrukkingen. Anders gezegd, de laatste binding van een variabele blijft gelden binnen de evaluatie waarin deze binding werd gemaakt. Bijvoorbeeld, als FOO opgeroepen wordt in een functie die wel een binding heeft voor Z, dan blijft die binding nog geldig. Dus, gegeven

```
(DEFUN BAR (Z) (FOO 5 6))
```

Dan is (BAR 10) gelijk aan 10, omdat de binding van Z en 10 gemaakt bij de oproep van BAR blijft gelden binnen FOO.

Dynamisch bereik wijkt enigszins af van de basisprincipes van applicatief programmeren omdat programma's niet langer modulair zijn. Daarom veronderstellen we in verdere oefeningen steeds dat we te maken hebben met een evaluator met een lexicaal bereik. Dit is overigens ook de COMMON LISP standaard.

Het oproepen van een functie is een manier om een aantal bindingen te realiseren. Soms is het handig om bindingen te realiseren binnen in een functie, bijvoorbeeld om deelresultaten te bewaren. Dit kan in LISP met een speciale functie LET met twee argumenten, een lijst van bindingen en een lijst van formules:

$$
\begin{array}{l}
\text{(LET ((variabele}_1\text{ uitdrukking}_1 \\
\quad\quad \text{(variabele}_2\text{ uitdrukking}_2\text{)} \\
\quad\quad \text{...)} \\
\quad \text{formule}_1\text{ formule}_2\text{ ...).}
\end{array}
$$

Bij het evalueren van deze uitdrukking worden de variabelen gebonden aan het resultaat van de evaluatie van de respectievelijke uitdrukkingen en dan worden de formules geëvalueerd.

Bijvoorbeeld, de evaluatie van

```
(LET ((X 1) (Y 2))
  (+ X Y))
```

bindt X aan 1 en Y aan 2 en voert dan (+ X Y) uit met als resultaat 3.

Merk op dat de bindingen "parallel" zijn in de zin dat vroegere bindingen nog niet gelden bij het tot stand brengen van de volgende bindingen. Dus

```
(LET ((X 15)
      (Y (* 2 X)))
  (- Y X))
```

is UNDEFINED omdat X nog niet gebonden is als we Y willen binden aan (* 2 X).

LET* is een variant van LET waar de bindingen *wel* sequentiëel worden uitgevoerd. Dus

```
(LET* ((X 15)
       (Y (* 2 X)))
  (- Y X))
```

is nu wel een goede manier om de bindingen tot stand te brengen. Het resultaat is 15.

Het bereik van de bindingen geïntroduceerd door LET en LET* hangt opnieuw af van het feit of het LISP-systeem lexicaal of dynamisch is opgezet. In een lexicale implementatie zijn de variabelen gebonden bij het oproepen van de functie in de formule. In een dynamische implementatie blijven de variabelen gebonden binnen in de evaluatie van de formule.

2.2. OEFENINGEN

1. Gegeven de volgende twee functies:

```
(DEFUN FOO (X Y)
  (BAR (+ X 1)))

(DEFUN BAR (X)
  Y)
```

---

2. Veronderstel een LISP met een dynamisch bereik, wat is de evaluatie van (FOO 1 2)?

2, omdat X gebonden wordt aan 1 en Y aan 2 in de oproep van FOO. In BAR wordt X gelijk aan 2, maar Y blijft zijn waarde behouden.

3. Veronderstel een LISP met een lexicaal bereik, wat is de evaluatie van (FOO 1 2)?

UNDEFINED, want alhoewel X aan 1 en Y aan 2 gebonden worden tijdens de oproep van FOO, krijgt alleen X een waarde in BAR.

4. Evalueer
```
(LET ((X (+ 5 10))
      (Y (+ 10 20)))
  (+ X Y))
```

45, X wordt gebonden aan 15 en Y aan 30.

5. Evalueer
```
(LET ((X (+ 15 20))
      (Y 30)
      (Z (+ X Y)))
  (+ X Y Z))
```

UNDEFINED, omdat de waarde van X en Y nog niet gekend zijn bij het berekenen van de binding voor Z.

6. Verbeter deze uitdrukking met behoud van de variabelen.

```
(LET* ((X (+ 15 20))
       (Y 30)
       (Z (+ X Y)))
  (+ X Y Z))
```

3. SUBSTITUTIEFUNCTIES

3.1. DEFINITIES

In dit en de komende onderdelen bekijken we enkele typische definitieschema's. Het eenvoudigste soort definities is gebaseerd op het idee van substitutie: er wordt abstractie gemaakt van een functie of een compositie van functies en deze abstractie wordt een naam gegeven. TWEEDE is een voorbeeld van een substitutiefunctie. TWEEDE is gelijk aan een compositie van CAR en CDR. De volgende oefeningen bevatten meer voorbeelden.

3.2. OEFENINGEN

1. Definieer een functie DERDE die het derde element van een lijst oplevert.

```
(DEFUN DERDE (L)
  (CAR (CDR (CDR L))))
```

2. Wat is de eerste stap in de evaluatie van (DERDE '(A B C))?

L wordt gebonden aan (A B C) en (CAR (CDR (CDR L))) wordt geëvalueerd.

3. Definieer een functie MAAK-LIJST die een lijst maakt van zijn twee argumenten. Bijvoorbeeld (MAAK-LIJST 'A 'B) is gelijk aan (A B). Gebruik CONS.

```
(DEFUN MAAK-LIJST (A B)
  (CONS A (CONS B NIL)))
```

4. Definieer een predikaat LISTP dat T oplevert als het argument een lijst is anders NIL. Bijvoorbeeld (LISTP '(A B C)) levert T op.

```
(DEFUN LISTP (L)
  (NOT (SYMBOLP L)))
```

5. Definieer een predikaat dat gelijk is aan T als een lijst slechts 1 element bevat.

```
(DEFUN SLECHTS-EEN-ELEMENT (LIJST)
  (AND LIJST (NULL (CDR LIJST))))
```

6. Schrijf een functie DUBBELE die het dubbele van een getal berekent.

```
(DEFUN DUBBELE (X)
  (+ X X))
```

7. Schrijf een functie XOR voor een exclusieve OR. Dus (XOR T T) is NIL, i.p.v. T zoals bij de inclusieve OR.

```
(DEFUN XOR (X Y)
  (OR (AND X (NULL Y))
      (AND (NULL X) Y)))
```

8. Schrijf een functie EVENP die T oplevert als een getal even is, anders NIL.

```
(DEFUN EVENP (GETAL)
  (= (* (/ GETAL 2) 2) GETAL))
```

Bijvoorbeeld, (/ 15 2) is 7 (/ rondt af), en 7 * 2 is 14. 14 is niet gelijk aan 15, dus is (EVENP 15) gelijk aan NIL. LISP heeft normaal een primitieve EVENP functie en ook een ODDP functie die nagaat of een getal oneven is.

4. RECURSIEVE FUNCTIES I: STAARTRECURSIE

4.1. DEFINITIES

Recursieve functies zijn functies die direct of indirect de te definiëren functie oproepen in de definitie zelf.

Een functie MEMBER die nagaat of een symbool een element is van een lijst kan bijvoorbeeld als volgt gedefinieerd worden:

1. Als de lijst gelijk is aan de lege lijst is het symbool er zeker niet in.
2. Als het symbool gelijk is aan het eerste element van de lijst is het er wel in.
3. Anders passen we MEMBER toe op de rest van de lijst.

In LISP:

```
(DEFUN MEMBER (SYMBOOL LIJST)
  (COND
    ((NULL LIJST) NIL)
    ((EQ (CAR LIJST) SYMBOOL) T)
    (T (MEMBER SYMBOOL (CDR LIJST)))))
```

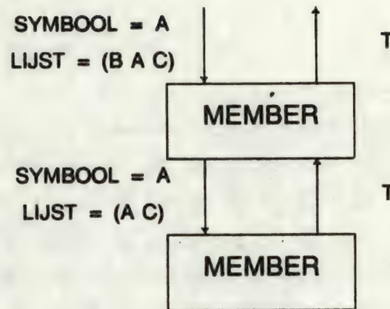
Een recursieve definitie bevat de volgende componenten:

1. Formules voor eenvoudige gevallen: bijvoorbeeld als de lijst gelijk is aan NIL, voor het eerste element van de lijst, enz.
2. Een generalisering die gebruik maakt van deelresultaten verkregen door de functie toe te passen op onderdelen van de oorspronkelijke argumenten.

Er is dus een reductieve fase waarin het probleem teruggebracht wordt tot de eenvoudige gevallen en een constructieve fase waarin de deelresultaten gecombineerd worden.

Het is uiteraard belangrijk dat er een eindconditie is zodanig dat de recursie stopt, en het is ook belangrijk dat het heroproepen van de functie altijd gebeurt met onderdelen van de argumenten, bijvoorbeeld de rest van de lijst. Als deze twee voorwaarden niet voldaan zijn zal de recursie oneindig blijven duren.

Om recursieve toepassingen van een functie na te gaan is het nuttig om een schematische representatie van het evaluatieproces in de vorm van een diagram te construeren. Het diagram bevat voor elke oproep de bindingen bij het oproepen van de functie en het resultaat na het verlaten van de functie. Bijvoorbeeld de evaluatie van (MEMBER 'A '(B A C)) leidt tot het volgende diagram:



Links staan de bindingen bij het binnenkomen in een functieoproep van MEMBER. Rechts staat het resultaat na het evalueren van deze oproep. In de eerste oproep is SYMBOOL gelijk aan A en LIJST gelijk aan (B A C). Omdat de lijst niet de null-lijst is en omdat het eerste element niet gelijk is aan A, is er een recursieve oproep van MEMBER. Dus we krijgen een tweede oproep van MEMBER met SYMBOOL gelijk aan A en LIJST gelijk aan (A C). Het eerste element is nu wel gelijk aan SYMBOOL en dus is het resultaat van deze oproep T. Dit resultaat wordt gewoon doorgegeven in de volgende oproep zodanig dat op het eerste niveau ook T voorkomt.

Via deze diagrammen kunnen we het gedrag van recursieve functies gemakkelijk bestuderen. We kunnen bijvoorbeeld zien of de argumenten wel degelijk korter (of kleiner) worden. Als dit niet het geval is zal de recursie niet stoppen. We kunnen ook nagaan wat er gebeurt met het resultaat van een niveau, voor het naar buiten komt in het hogere niveau.

4.2. OEFENINGEN

1. Gegeven de formule (MEMBER 'A '(B A C)), wat is de eerste stap in de evaluatie?

   SYMBOOL wordt gebonden aan A en LIJST wordt gebonden aan (B A C) en de definitie van MEMBER,

```
(COND
  ((NULL LIJST) NIL)
  ((EQ (CAR LIJST) SYMBOOL) T)
  (T (MEMBER SYMBOOL (CDR LIJST)))),
```

wordt geëvalueerd.

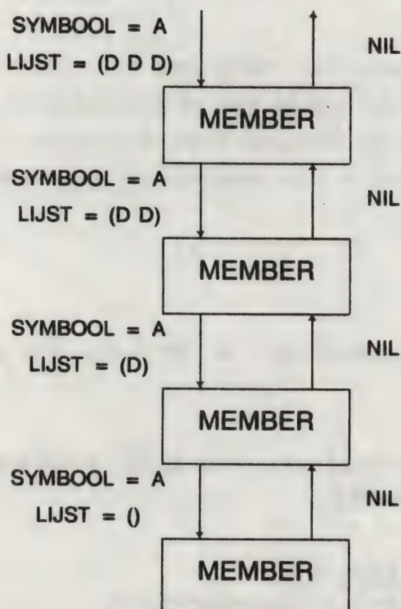
2. Wat is de volgende stap?

De voorwaardelijke uitdrukking wordt toegepast. Dit wil zeggen dat het eerste predikaat, NULL, wordt toegepast op (B A C). Het levert NIL op. Het tweede predikaat gaat na of A gelijk is aan B. A is verschillend van B dus dit levert ook NIL op. Het derde predikaat T is altijd waar zodanig dat nu (MEMBER SYMBOOL (CDR LIJST)) geëvalueerd moet worden met SYMBOOL gelijk aan A en (CDR LIJST) gelijk aan (A C).

3. Wat is de volgende stap?

MEMBER wordt opnieuw toegepast. SYMBOOL wordt gebonden aan A, LIJST wordt gebonden aan (A C) en de definitie wordt uitgevoerd.

4. Teken het diagram voor (MEMBER 'A '(D D D))



5. Veronderstel een functie SYMBOL-TEST die T oplevert als alle elementen in een lijst symbolen zijn, anders NIL. Gegeven L gelijk aan (A B C), wat is (SYMBOL-TEST L)?

T, want A, B en C zijn symbolen.

6. Gegeven L gelijk aan ((A) B C), wat is (SYMBOL-TEST L)?

NIL, want (A) is geen symbool.

7. Gegeven L gelijk aan NIL, wat is (SYMBOL-TEST L)?

T, want alle elementen van L zijn symbolen. Er zijn geen elementen dus het predikaat is altijd waar.

8. Schrijf een voorwaardelijke uitdrukking die test of de elementen van L symbolen zijn. Het is toegelaten te veronderstellen dat SYMBOL-TEST reeds gedefinieerd is.

```
(COND
  ((NULL L) T)
  ((SYMBOLP (CAR L)) (SYMBOL-TEST (CDR L)))
  (T NIL))
```

9. Definieer de functie SYMBOL-TEST.

```
(DEFUN SYMBOL-TEST (L)
  (COND
    ((NULL L) T)
    ((SYMBOLP (CAR L)) (SYMBOL-TEST (CDR L)))
    (T NIL)))
```

10. Veronderstel dat M gelijk is aan (B C), wat gebeurt er eerst voor (SYMBOL-TEST M)?

L wordt gelijk gesteld aan de waarde van M, en (SYMBOL-TEST M) wordt vervangen door

```
(COND
  ((NULL L) T)
  ((SYMBOLP (CAR L)) (SYMBOL-TEST (CDR L)))
  (T NIL))
```

11. Wat gebeurt er dan?

Het eerste predikaat in de voorwaardelijke uitdrukking wordt getest: (NULL L). L is gelijk aan (B C) dus (NULL L) levert NIL op.

12. Wat gebeurt er dan?

Het tweede predikaat in de voorwaardelijke uitdrukking wordt getest: (SYMBOLP (CAR L)). (CAR L) is gelijk aan B en dus (SYMBOLP L) is T want B is een symbool. Omdat dit predikaat waar is, is het resultaat van de uitdrukking gelijk aan de waarde van (SYMBOL-TEST (CDR L)).

13. Wat is de volgende stap?

De functiedefinitie van SYMBOL-TEST wordt opnieuw gebruikt. L wordt gelijk gezet aan (CDR L), d.w.z. (C), en

```
(COND
  ((NULL L) T)
  ((SYMBOLP (CAR L)) (SYMBOL-TEST (CDR L)))
  (T NIL))
```

wordt geëvalueerd.

14. Wat is de volgende stap?

(NULL L) levert NIL op en (SYMBOLP (CAR L)) levert T op omdat C een symbool is. Het resultaat is nu gelijk aan (SYMBOL-TEST (CDR L)).

15. Wat is de laatste stap?

SYMBOL-TEST wordt opnieuw toegepast. L is nu gelijk aan (CDR L), d.w.z. NIL, en (SYMBOL-TEST (CDR L)) wordt vervangen door

```
(COND
 ((NULL L) T)
 ((SYMBOLP (CAR L)) (SYMBOL-TEST (CDR L)))
 (T NIL))
```

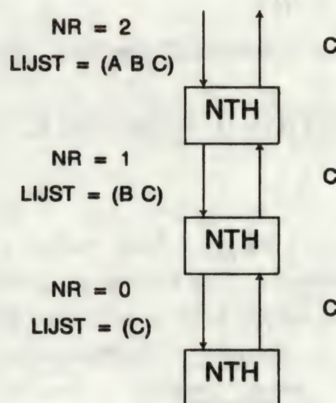
Het eerste predikaat, (NULL L), levert T op omdat L de nullijst is. Dus het eindresultaat is T.

16. Definieer een functie NTH die het $(n+1)^e$ element van een lijst geeft. 0 is het eerste element. Als n groter is dan de lengte van de lijst is het resultaat NIL. Bijvoorbeeld, (NTH 0 '(A B C)) is gelijk aan A omdat A het eerste element van de lijst is.

Analyse: Als de lijst gelijk is aan NIL, is het resultaat NIL omdat er geen n^e element is. Als n gelijk is aan 0 is het resultaat gelijk aan het eerste element van de lijst. De rest van het probleem kan gereduceerd worden tot één van deze twee gevallen door te veronderstellen dat het $(n-1)^e$ element moet gezocht worden in de rest van de lijst. In LISP:

```
(DEFUN NTH (NR LIJST)
  (COND
   ((NULL LIJST) NIL)
   ((= NR 0) (CAR LIJST))
   (T (NTH (- NR 1) (CDR LIJST)))))
```

17. Construeer een diagram voor (NTH 2 '(A B C)).



18. Definieer een functie LAATSTE-ELEMENT die het laatste element van een lijst oplevert. Bijvoorbeeld (LAATSTE-ELEMENT '(A B C)) is gelijk aan C.

Analyse: Als de lijst gelijk is aan de lege lijst is het resultaat duidelijk gelijk aan NIL. Als de lijst slechts één element bevat is dit het element dat we nodig hebben. Anders is het probleem gelijk aan het laatste element van de rest van de lijst.

```
(DEFUN LAATSTE-ELEMENT (L)
  (COND
    ((NULL L) NIL)
    ((NULL (CDR L)) (CAR L))
    (T (LAATSTE-ELEMENT (CDR L)))))
```

19. Definieer de functie > die T oplevert als het eerste argument groter is dan het tweede argument, anders NIL.

Analyse: Als de argumenten gelijk zijn aan elkaar is het antwoord NIL. Als het eerste argument gelijk is aan 0 is het antwoord ook NIL. Als het tweede argument gelijk is aan 0 is het antwoord T. Als geen van deze voorwaarden voldaan is, kan het probleem verder gereduceerd worden door te kijken of het eerste argument groter is dan het tweede argument min 1.

```
(DEFUN > (X Y)
  (COND
    ((= X Y) NIL)
    ((= X 0) NIL)
    ((= Y 0) T)
    (T (> X (- Y 1)))))
```

Merk op dat deze definitie veronderstelt dat beide getallen groter zijn dan 0.

5. RECURSIEVE FUNCTIES II: CONSTRUCTIEVE RECURSIE

5.1. DEFINITIES

De recursieve functies die we tot hiertoe hebben gezien, geven telkens het resultaat door zonder er iets aan te wijzigen. We kunnen dit goed zien in de diagrammen. Het resultaat dat uit de laatste oproep komt is gelijk aan het eindresultaat. Dit heet *staartrecursie* en kan leiden tot belangrijke optimaliseringen in het evaluatieproces en in gecompileerde code. Meer bepaald is het niet echt nodig om een

stapelgeheugen te gebruiken om de deelresultaten van een recursieve stap te bewaren. Er moet daarom een onderscheid gemaakt worden tussen een recursieve definitie (de functie roept direct of indirect zichzelf op) en een recursief proces (een stapelgeheugen bewaart de deelresultaten van de recursieve oproep omdat die later nog nodig zijn). Staartrecursie is een recursieve definitie van een iteratief proces.

Constructieve recursie is recursie waar wel degelijk wijzigingen aan het resultaat van de recursieve toepassing van de functie aangebracht worden en waar dus een stapelgeheugen nodig is om deeltijdse resultaten te bewaren. Constructieve recursie is niet alleen een recursieve definitie, maar beschrijft ook een recursief proces.

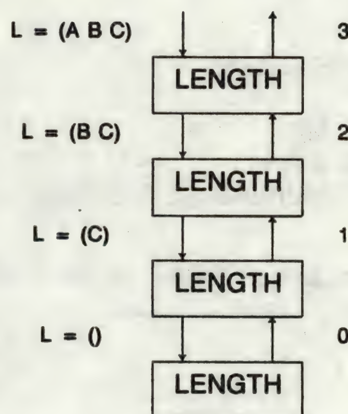
Een functie om de lengte van een lijst te berekenen kan bijvoorbeeld als volgt gedefinieerd worden:

1. Als de lijst gelijk is aan de lege lijst, is de lengte gelijk aan 0.
2. Anders is de lengte van de lijst gelijk aan 1 + de lengte van de rest van de lijst.

In LISP:

```
(DEFUN LENGTH (L)
  (COND
    ((NULL L) 0)
    (T (+ 1 (LENGTH (CDR L))))))
```

Het diagram voor (LENGTH '(A B C)) ziet er als volgt uit:



Merk op hoe de uitkomst van LENGTH telkens gewijzigd wordt.

5.2. OEFENINGEN

1. Definieer een functie REEKS-SOM die een som maakt van de elementen in een lijst. Bijvoorbeeld (REEKS-SOM '(1 2 3 4)) is gelijk aan 10.

```
(DEFUN REEKS-SOM (ARGUMENTEN)
  (COND
    ((NULL ARGUMENTEN) 0)
    (T (+ (CAR ARGUMENTEN)
          (REEKS-SOM (CDR ARGUMENTEN))))))
```

2. We gaan een functie APPEND schrijven die twee lijsten tot een lijst samensmelt. Bijvoorbeeld, (APPEND '(A B C) '(D E F)) is gelijk aan (A B C D E F). Wat is het eenvoudigste geval?

   Dat de eerste lijst gelijk is aan NIL, want dan is het resultaat gelijk aan de tweede lijst.

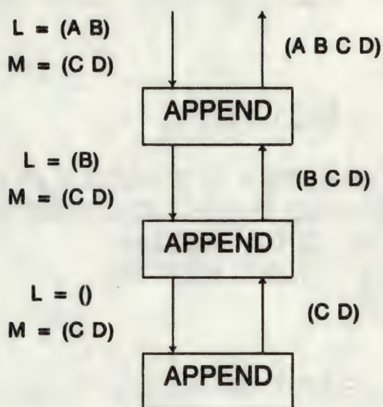
3. Wat is de eenvoudigste constructieve stap?

   Eén enkel element toevoegen aan de andere lijst.

4. Schrijf een definitie van APPEND.

```
(DEFUN APPEND (L M)
  (COND
    ((NULL L) M)
    (T (CONS (CAR L)
             (APPEND (CDR L) M)))))
```

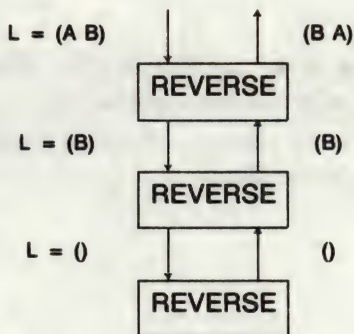
5. Construeer het diagram voor (APPEND '(A B) '(C D))



6. Schrijf nu een functie REVERSE, die het omgekeerde van een lijst construeert. Bijvoorbeeld, (REVERSE '(A B C)) is gelijk aan (C B A). Hint: gebruik APPEND voor de constructieve fase.

```
(DEFUN REVERSE (L)
  (COND
    ((NULL L) NIL)
    (T (APPEND (REVERSE (CDR L))
               (CONS (CAR L) NIL)))))
```

7. Construeer het diagram voor (REVERSE '(A B)).



8. Definieer een functie EQUAL die nagaat of een lijst gelijk is aan een andere lijst.

———

```
(DEFUN EQUAL (L M)
  (COND
    ((AND (SYMBOLP L) (SYMBOLP M) (EQ L M)) T)
    ((OR (SYMBOLP L) (SYMBOLP M)) NIL)
    ((EQUAL (CAR L) (CAR M))
      (EQUAL (CDR L) (CDR M)))
    (T NIL)))
```

9. Schrijf een functie SUBST die een LISP-object vervangt door een andere LISP-object in een lijst. Bijvoorbeeld, (SUBST 'A 'B '(A B C)) levert (A A C) op. Gebruik EQUAL om de elementen te vergelijken.

———

```
(DEFUN SUBST (SUBSTITUUT ORIGINEEL LIJST)
  (COND
    ((NULL LIJST) NIL)
    ((EQUAL (CAR LIJST) ORIGINEEL)
      (CONS SUBSTITUUT
                (SUBST SUBSTITUUT ORIGINEEL (CDR LIJST))))
    (T
      (CONS
        (CAR LIJST)
        (SUBST SUBSTITUUT ORIGINEEL (CDR LIJST))))))
```

10. Definieer een functie TEL-SYMBOLEN die het aantal symbolen in een lijst S telt.

———

Analyse: Er zijn drie gevallen: (i) S is gelijk aan de lege lijst, dan is het resultaat 0, (ii) S is gelijk aan een symbool, dan is het resultaat 1, (iii) anders moeten we het aantal symbolen in het eerste element optellen bij het aantal symbolen in de rest van de lijst:

```
(DEFUN TEL-SYMBOLEN (S)
  (COND
    ((NULL S) 0)
    ((SYMBOLP S) 1)
    (T
      (+
        (TEL-SYMBOLEN (CAR S))
        (TEL-SYMBOLEN (CDR S))))))
```

Dit is een voorbeeld van *boomrecursie* omdat de te definiëren functie zichzelf meer dan eens oproept.

11. Schrijf een functie FLATTEN die alle haakjes verwijdert uit een lijst. Bijvoorbeeld (FLATTEN '(A (B) ((C)))) is gelijk aan '(A B C).

Analyse: Er zijn drie gevallen. Ofwel is de lijst leeg dan is het resultaat NIL. Ofwel is het eerste element een symbool, en dan kunnen we het zo toevoegen. Ofwel is het eerste element geen symbool en dan passen we FLATTEN toe op het eerste element en voegen dit toe via APPEND aan het resultaat van FLATTEN toegepast op de rest van de lijst.

```
(DEFUN FLATTEN (LIJST)
  (COND
    ((NULL LIJST) NIL)
    ((SYMBOLP (CAR LIJST))
      (CONS
        (CAR LIJST)
        (FLATTEN (CDR LIJST))))
    (T
      (APPEND
        (FLATTEN (CAR LIJST))
        (FLATTEN (CDR LIJST))))))
```

FLATTEN is opnieuw een voorbeeld van boomrecursie.

6. RECURSIEVE FUNCTIES III: FUNCTIES MET ACCUMULATOREN

6.1. DEFINITIES

Het is soms nodig extra variabelen toe te voegen aan een recursief gedefinieerde functie. Deze variabelen functioneren als geheugen waarin informatie bewaard wordt die nodig is bij recursieve oproepen. We noemen deze extra variabelen accumulatoren. Recursieve functies die extra geheugen nodig hebben noemen we recursieve functies met accumulatoren. De accumulatoren worden geïnitialiseerd bij het oproepen van de functie.

Veronderstel bijvoorbeeld dat we een functie VERSCHILLENDE-ELEMENTEN nodig hebben die nagaat hoeveel *verschillende* elementen er in een lijst zijn. Het probleem is dat we niet kunnen veronderstellen dat de resultaten van het toepassen van de functie op een onderdeel zomaar gecombineerd worden met resultaten voor een kleiner onderdeel. In de plaats daarvan hebben we een soort geheugen nodig dat gebruikt kan worden in recursieve oproepen van VERSCHILLENDE-ELEMENTEN om de elementen die reeds aanwezig waren te onthouden. Als een nieuw element wordt ontmoet, voegen we het bij het geheugen en voegen we 1 toe bij de recursieve toepassing van VERSCHILLENDE-ELEMENTEN:

```
(DEFUN VERSCHILLENDE-ELEMENTEN (LIJST GEHEUGEN)
  (COND
    ((NULL LIJST) 0)
    ((MEMBER (CAR LIJST) GEHEUGEN)
      (VERSCHILLENDE-ELEMENTEN
        (CDR LIJST) GEHEUGEN))
    (T
      (+ 1
          (VERSCHILLENDE-ELEMENTEN
            (CDR LIJST) (CONS (CAR LIJST) GEHEUGEN))))))
```

Accumulatoren zijn nodig als het niet mogelijk is om een zuivere recursieve functie te schrijven. Accumulatoren kunnen ook gebruikt worden om een constructieve recursie om te zetten in staartrecursie. Bijvoorbeeld de definitie van REEKS-SOM met constructieve recursie ziet er als volgt uit (zie oefening 1 op pagina 2.18):

```
(DEFUN REEKS-SOM (ARGUMENTEN)
  (COND
    ((NULL ARGUMENTEN) 0)
    (T (+ (CAR ARGUMENTEN)
          (REEKS-SOM (CDR ARGUMENTEN))))))
```

maar ze kan ook zo geschreven worden:

```
(DEFUN REEKS-SOM (ARGUMENTEN)
  (AUX-REEKS-SOM ARGUMENTEN 0))

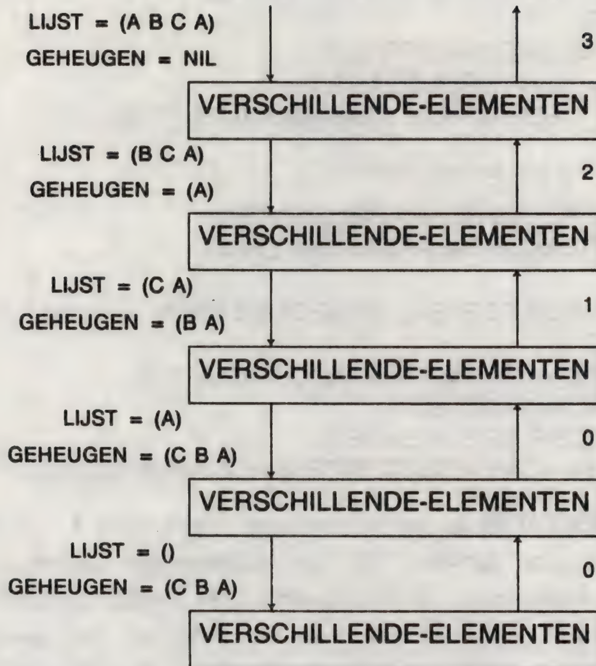
(DEFUN AUX-REEKS-SOM (ARGUMENTEN ACCUMULATOR)
  (COND
    ((NULL ARGUMENTEN) ACCUMULATOR)
    (T (AUX-REEKS-SOM
         (CDR ARGUMENTEN)
         (+ (CAR ARGUMENTEN) ACCUMULATOR)))))
```

Merk op dat REEKS-SOM nu via staartrecursie is gedefinieerd - en dus efficiënter zal zijn. Merk ook op dat deze stijl van programmeren nauwer aansluit bij de imperatieve programmeerstijl. In feite gedraagt AUX-REEKS-SOM zich als een DO-lus. De eerste oproep initialiseert de variabelen en elke oproep brengt de nodige wijzigingen aan. De eerste conditie van AUX-REEKS-SOM dient als eindconditie van de lus.

6.2. OEFENINGEN

1. Construeer een diagram voor (VERSCHILLENDE-ELEMENTEN '(A B C A) NIL).

———



2. Schrijf een nieuwe definitie voor REVERSE gebaseerd op recursie met accumulatoren. De functie heeft twee argumenten: de lijst en de accumulator. De functie wordt opgeroepen met de accumulator gelijk aan NIL.

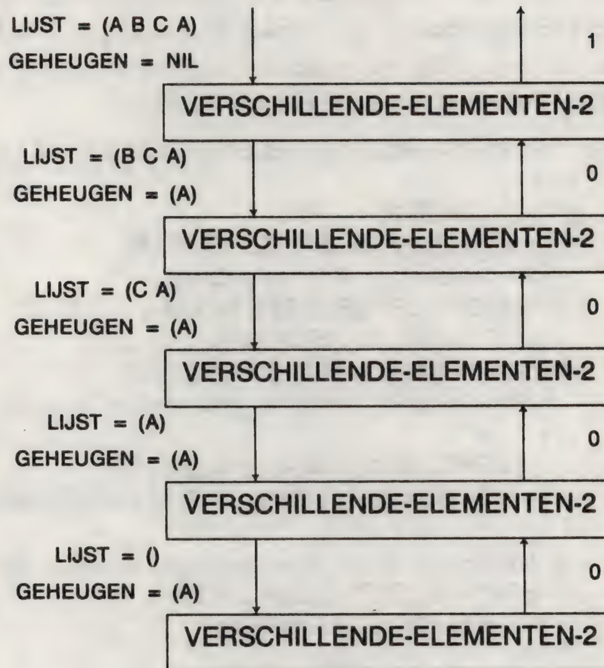
```
(DEFUN REVERSE (LIJST ACCUMULATOR)
  (COND
    ((NULL LIJST) ACCUMULATOR)
    (T
      (REVERSE
        (CDR LIJST)
        (CONS (CAR LIJST) ACCUMULATOR)))))
```

3. Definieer een functie VERSCHILLENDE-ELEMENTEN-2 die het aantal elementen in een lijst telt die meer dan één keer voorkomen. Bijvoorbeeld toegepast op '(A B C A) levert de functie 1 op want alleen A komt meer dan één keer voor.

Deze functie heeft dezelfde structuur als VERSCHILLENDE-ELEMENTEN. Een GEHEUGEN onthoudt of een element al aanwezig is. In de plaats van 1 bij het resultaat op te tellen van zodra het nog niet voorkomt, tellen we 1 op als het element nog voorkomt in de rest van de lijst.

```
(DEFUN VERSCHILLENDE-ELEMENTEN-2 (LIJST GEHEUGEN)
  (COND
    ((NULL LIJST) 0)
    ((MEMBER (CAR LIJST) GEHEUGEN)
      ;; Het element zit al in het geheugen
      (VERSCHILLENDE-ELEMENTEN-2
        (CDR LIJST) GEHEUGEN))
    ((MEMBER (CAR LIJST) (CDR LIJST))
      ;; Het element is niet in het geheugen en komt nog voor
      (+ 1
          (VERSCHILLENDE-ELEMENTEN-2
            (CDR LIJST) (CONS (CAR LIJST) GEHEUGEN))))
    (T
      ;; het element is niet in het geheugen en komt niet meer voor
      (VERSCHILLENDE-ELEMENTEN-2
        (CDR LIJST) GEHEUGEN))))
```

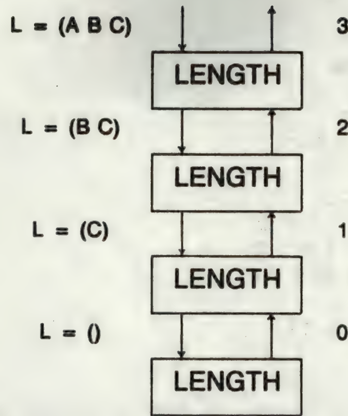
4. Construeer een diagram voor (VERSCHILLENDE-ELEMENTEN-2 '(A B C A) NIL).



7. VERSLAG

We hebben tot hiertoe diagrammen gebruikt om te onderzoeken hoe het evaluatieproces verloopt. In dit gedeelte bekijken we een andere manier: een verslag. Een verslag bevat voor elke functie waarin we geïnteresseerd zijn de manier waarop de functie wordt opgeroepen en het resultaat.

Veronderstel bijvoorbeeld dat we een verslag willen maken van het oproepen van de functie LENGTH voor (LENGTH '(A B C)). Hier is het diagram:



In de vorm van een verslag schrijven we dit als:

```
1. ENTER LENGTH
       L = (A B C)
  2. ENTER LENGTH
        L = (B C)
    3. ENTER LENGTH
          L = (C)
      4. ENTER LENGTH
            L = NIL
      4. EXIT LENGTH 0.
    3. EXIT LENGTH 1.
  2. EXIT LENGTH 2.
1. EXIT LENGTH 3.

Eindresultaat: 3.
```

LISP systemen kunnen gewoonlijk verslagen zoals deze zelf construeren. Zulke verslagen zijn een belangrijk hulpmiddel bij het opsporen van fouten.

7.1. OEFENINGEN

1. Construeer het verslag voor (APPEND '(A B) '(C D))

```
1. ENTER APPEND
      L = (A B)
      M = (C D)
  2. ENTER APPEND
        L = (B)
        M = (C D)
    3. ENTER APPEND
          L = NIL
          M = (C D)
    3. EXIT APPEND (C D)
  2. EXIT APPEND (B C D)
1. EXIT APPEND (A B C D)

Eindresultaat: (A B C D)
```

2. We kunnen uiteraard van meer dan één functie een verslag maken. Construee. het verslag voor REVERSE (zoals gedefinieerd op pagina 2.19) *en* APPEND, gegeven (REVERSE '(A B))

```
1. ENTER REVERSE
      L = (A B)
  2. ENTER REVERSE
        L = (B)
    3. ENTER REVERSE
          L = NIL
    3. EXIT REVERSE NIL
    1. ENTER APPEND
          L = NIL
          M = (B)
    1. EXIT APPEND (B)
  2. EXIT REVERSE (B)
  1. ENTER APPEND
        L = (B)
        M = (A)
    2. ENTER APPEND
        L = NIL
        M = (A)
    2. EXIT APPEND (A)
  1. EXIT APPEND (B A)
1. EXIT REVERSE (B A)

Eindresultaat: (B A)
```

3. Construeer een verslag voor (EQUAL '(A B) '(A C)).

```
1. ENTER EQUAL
      L = (A B)
      M = (A C)
  2. ENTER EQUAL
      L = A
      M = A
  2. EXIT EQUAL T
  2. ENTER EQUAL
      L = (B)
      M = (C)
    3. ENTER EQUAL
        L = B
        M = C
    3. EXIT EQUAL NIL
  2. EXIT EQUAL NIL
1. EXIT EQUAL NIL

Eindresultaat: NIL
```

8. SAMENVATTING

In dit deel hebben we leren functies definiëren met de functie DEFUN. De volgende definitieschema's werden bestudeerd:

1. Substitutiefuncties: de definitie is een compositie van andere functies.
2. Recursieve functies: de functie komt zelf voor in de definitie.
   a. Staartrecursie: er verandert niets aan het resultaat van de recursie.
   b. Constructieve recursie: er verandert wel iets aan het resultaat van de recursie.
   c. Recursie met accumulatoren: er is een extra geheugen nodig om globale eigenschappen te bewaren.

Bij het definiëren van een functie is het in de eerste plaats belangrijk om te zien welk definitieschema toegepast moet worden. De volgende stap is het analyseren van de gevallen.

In de oefeningen hebben we nog een aantal functies gezien die normaal aanwezig zijn als primitieve functies in een LISP systeem en die daarom behoren tot het basisrepertorium:

NTH levert het n-de element van een lijst. n is het eerste argument en de lijst is het tweede argument.

MEMBER is een predikaat dat nagaat of het eerste argument aanwezig is in het tweede argument. EQUAL is een predikaat dat nagaat of twee lijsten of symbolen

gelijk zijn.

APPEND construeert een nieuwe lijst van zijn twee argumenten. REVERSE levert het omgekeerde van een lijst. SUBST vervangt het tweede argument door het eerste argument in een lijst.

EVENP en ODDP gaan na of een getal even of oneven is.

9. GEMENGDE OPGAVEN

Met de functies die we tot hiertoe gezien hebben kan men al ontzettend veel doen. Bij wijze van voorbeeld behandelt dit hoofdstuk een aantal "echte" problemen die de lezer kan gebruiken om zijn kennis te testen en uit te diepen. Eerst worden de opgaven beschreven. Dan worden mogelijke oplosssingen besproken. De bedoeling is uiteraard dat de lezer eerst probeert zelf de problemen op te lossen alvorens de oplossingen te bestuderen. Anderzijds kan hij het volgende hoofdstuk beginnen en later deze opgaven trachten op te lossen. Een laatste paragraaf bevat nog enkele opgaven die niet verder worden opgelost.

9.1. DE OPGAVEN

9.1.1. DE TORENS VAN HANOI.

Volgens de legende is er een tempel in het Verre Oosten waar monniken zich bezig houden met het verplaatsen van 64 schijven van verschillende grootte op 3 pennen. De monniken mogen slechts één schijf per keer verplaatsen en een grotere schijf mag nooit op een kleinere liggen. In het begin staan alle schijven op de eerste pen met de grootste schijf onderaan. De bedoeling is alle schijven op dezelfde manier gerangschikt op de derde pen te krijgen. Volgens de legende is het einde van de wereld nabij als de monniken dit punt bereikt hebben.

Schrijf een programma dat de monniken vertelt hoe ze te werk moeten gaan. Het doet er niet toe hoeveel schijven gebruikt worden. Hier is een voorbeeld van de instructies voor 3 schijven.

```
((VERPLAATS SCHIJF 1. VAN PEN1 NAAR PEN3)
 (VERPLAATS SCHIJF 2. VAN PEN1 NAAR PEN2)
 (VERPLAATS SCHIJF 1. VAN PEN3 NAAR PEN2)
 (VERPLAATS SCHIJF 3. VAN PEN1 NAAR PEN3)
 (VERPLAATS SCHIJF 1. VAN PEN2 NAAR PEN1)
 (VERPLAATS SCHIJF 2. VAN PEN2 NAAR PEN3)
 (VERPLAATS SCHIJF 1. VAN PEN1 NAAR PEN3))
```

9.1.2. ALGEBRAISCHE VEREENVOUDIGING

Algebraïsche uitdrukkingen kunnen worden gerepresenteerd door lijsten, zoals (+ A B), (+ (* 5 2) B), enz. We willen deze uitdrukkingen vereenvoudigen volgens de regels van de algebra. Er zijn twee typen van regels:

1. Eliminatieregels, bijvoorbeeld (+ A 0) is gelijk aan A, (/ B 1) is B, (* 0 A) is 0, enz.
2. Toepassing van een functie: Als de termen constant zijn, wordt de functie toegepast. Bijvoorbeeld (+ 3 4) is gelijk aan 7.

Uiteraard moeten deze regels recursief gebruikt worden. Bijvoorbeeld (+ (+ 10 2) 0) is gelijk aan (+ 10 2) (door eliminatie van 0) en verder gelijk aan 12 (toepassen van + op de constanten).

Schrijf een functie die deze vereenvoudigingen uitvoert.

9.1.3. VAN INFIX NAAR PREFIX

Een van de dingen waar veel beginners last mee hebben is de prefixnotatie van LISP. Dit is vlug verholpen met een programma dat een rekenkundige uitdrukking in infixnotatie omzet in prefixnotatie. Bijvoorbeeld (5 + 2 * 3) wordt omgezet in (+ 5 (* 2 3)). De hiërarchie van de operatoren is ↑, / , *, -, +. Schrijf een programma dat deze omzetting uitvoert.

9.2. OPLOSSINGEN

9.2.1. DE TORENS VAN HANOI

Er zijn drie pennen en de schijven moeten van de eerste pen naar de derde. De tweede pen kan dus als tussentijdse opslagplaats gebruikt worden. Als we n-1 schijven van de eerste pen naar de tussentijdse opslagpen kunnen krijgen, dan kan de laatste schijf naar het einddoel en is het probleem gereduceerd tot het brengen van n-1 schijven van de opslagpen naar het einddoel met de eerste pen als tussentijdse opslagpen. De recursiviteit stopt voor n = 1, want dan kan de schijf eenvoudigweg naar het einddoel overgeplaatst worden.

We schrijven dus een functie VERPLAATS-REEKS die een aantal schijven van een oorsprong naar een doel brengt, gegeven een pen als tussentijdse opslagpen. Veronderstel een functie VERPLAATS-SCHIJF die één enkele schijf brengt van een oorsprong naar een doel. Er zijn twee mogelijkheden:

1. Als het aantal schijven gelijk is aan 1, dan kunnen we de schijf gewoon van de oorsprong naar het doel verschuiven.

2. Als het aantal schijven groter is dan 1, dan moet het probleem gereduceerd worden tot drie stappen: (i) het verplaatsen van n-1 schijven van de oorsprong naar de opslagpen, (ii) het verplaatsen van de resterende schijf naar het einddoel, en (iii) het verplaatsen van n-1 schijven van de opslagpen naar het einddoel met de oorsprongpen als opslagpen:

```
(DEFUN VERPLAATS-REEKS (OORSPRONG DOEL
                                  OPSLAG AANTAL-SCHIJVEN)
  (COND
   ((= AANTAL-SCHIJVEN 1)
    (VERPLAATS-SCHIJF OORSPRONG DOEL))
   (T
    (APPEND
     (VERPLAATS-REEKS
      OORSPRONG OPSLAG
      DOEL (- AANTAL-SCHIJVEN 1))
     (VERPLAATS-SCHIJF OORSPRONG DOEL)
      (VERPLAATS-REEKS
       OPSLAG DOEL
       OORSPRONG (- AANTAL-SCHIJVEN 1))))))
```

VERPLAATS-SCHIJF ziet er zo uit:

```
(DEFUN VERPLAATS-SCHIJF (OORSPRONG DOEL)
  (LIST
   '(VERPLAATS SCHIJF VAN ,OORSPRONG NAAR ,DOEL)))
```

en het geheel begint als volgt:

```
(DEFUN TOREN-VAN-HANOI (AANTAL-SCHIJVEN)
  (VERPLAATS-REEKS 'PEN1 'PEN3 'PEN2 AANTAL-SCHIJVEN))
```

Om alles duidelijker te maken kunnen we de schijven nummeren en expliciet vermelden welke schijf verplaatst moet worden. Dit kan door VERPLAATS-SCHIJF een extra argument te geven:

```
(DEFUN VERPLAATS-SCHIJF (OORSPRONG DOEL SCHIJF)
  (LIST
   '(VERPLAATS SCHIJF ,SCHIJF
     VAN ,OORSPRONG NAAR ,DOEL)))
```

VERPLAATS-REEKS wordt dan

```
(DEFUN VERPLAATS-REEKS
  (OORSPRONG DOEL OPSLAG AANTAL-SCHIJVEN)
  (COND
   ((= AANTAL-SCHIJVEN 1)
    (VERPLAATS-SCHIJF OORSPRONG DOEL 1))
   (T
    (APPEND
     (VERPLAATS-REEKS
      OORSPRONG OPSLAG DOEL (- AANTAL-SCHIJVEN 1))
     (VERPLAATS-SCHIJF
      OORSPRONG DOEL AANTAL-SCHIJVEN)
     (VERPLAATS-REEKS
      OPSLAG DOEL OORSPRONG (- AANTAL-SCHIJVEN 1))))))
```

Hier is het resultaat voor (TOREN-VAN-HANOI 4):

```
((VERPLAATS SCHIJF 1. VAN PEN1 NAAR PEN2)
 (VERPLAATS SCHIJF 2. VAN PEN1 NAAR PEN3)
 (VERPLAATS SCHIJF 1. VAN PEN2 NAAR PEN3)
 (VERPLAATS SCHIJF 3. VAN PEN1 NAAR PEN2)
 (VERPLAATS SCHIJF 1. VAN PEN3 NAAR PEN1)
 (VERPLAATS SCHIJF 2. VAN PEN3 NAAR PEN2)
 (VERPLAATS SCHIJF 1. VAN PEN1 NAAR PEN2)
 (VERPLAATS SCHIJF 4. VAN PEN1 NAAR PEN3)
 (VERPLAATS SCHIJF 1. VAN PEN2 NAAR PEN3)
 (VERPLAATS SCHIJF 2. VAN PEN2 NAAR PEN1)
 (VERPLAATS SCHIJF 1. VAN PEN3 NAAR PEN1)
 (VERPLAATS SCHIJF 3. VAN PEN2 NAAR PEN3)
 (VERPLAATS SCHIJF 1. VAN PEN1 NAAR PEN2)
 (VERPLAATS SCHIJF 2. VAN PEN1 NAAR PEN3)
 (VERPLAATS SCHIJF 1. VAN PEN2 NAAR PEN3))
```

9.2.2. ALGEBRAISCHE VEREENVOUDIGING

De vereenvoudiging van een uitdrukking omvat drie gevallen:

1. De uitdrukking is een symbool; in dit geval is de vereenvoudiging gelijk aan het symbool zelf.
2. De uitdrukking is een som, verschil, produkt of deling; in dit geval passen we de regels voor vereenvoudiging toe geassocieerd met deze functies.

3. De uitdrukking is geen van beide; in dit geval blijft ze zoals ze is.

We schrijven nu de functie VEREENVOUDIG, waarbij we veronderstellen dat er andere functies zijn V-SOM, V-VERSCHIL, enz, die de uitdrukking behandelen voor het tweede geval.

```
(DEFUN VEREENVOUDIG (UITDRUKKING)
 (COND
  ((SYMBOLP UITDRUKKING) UITDRUKKING)
  ((EQ (CAR UITDRUKKING) '+)
   (V-SOM
    (VEREENVOUDIG (CADR UITDRUKKING))
    (VEREENVOUDIG
     (CADDR UITDRUKKING))))
  ((EQ (CAR UITDRUKKING) '-)
   (V-VERSCHIL
    (VEREENVOUDIG (CADR UITDRUKKING))
    (VEREENVOUDIG (CADDR UITDRUKKING))))
  ((EQ (CAR UITDRUKKING) '*)
   (V-PRODUKT
    (VEREENVOUDIG (CADR UITDRUKKING))
    (VEREENVOUDIG (CADDR UITDRUKKING))))
  ((EQ (CAR UITDRUKKING) '/)
   (V-DELING
    (VEREENVOUDIG (CADR UITDRUKKING))
    (VEREENVOUDIG (CADDR UITDRUKKING))))
  (T UITDRUKKING)))
```

Merk op hoe de functie VEREENVOUDIG recursief wordt opgeroepen voor de argumenten van de uitdrukking.

Nu een functie V-SOM die de vereenvoudiging van een som uitvoert:

```
(DEFUN V-SOM (ARGUMENT1 ARGUMENT2)
 (COND
  ((AND
    (NUMBERP ARGUMENT1)
    (NUMBERP ARGUMENT2))
   (+ ARGUMENT1 ARGUMENT2))
  ((= ARGUMENT1 0) ARGUMENT2)
  ((= ARGUMENT2 0) ARGUMENT1)
  (T (LIST '+ ARGUMENT1 ARGUMENT2))))
```

Een functie V-VERSCHIL die de vereenvoudiging van een verschil uitvoert:

```
(DEFUN V-VERSCHIL (ARGUMENT1 ARGUMENT2)
  (COND
    ((AND
       (NUMBERP ARGUMENT1)
       (NUMBERP ARGUMENT2))
     (- ARGUMENT1 ARGUMENT2))
    ((= ARGUMENT1 0) ARGUMENT2)
    ((= ARGUMENT2 0) ARGUMENT1)
    (T (LIST '- ARGUMENT1 ARGUMENT2))))
```

Een functie V-PRODUKT die de vereenvoudiging van een produkt uitvoert:

```
(DEFUN V-PRODUKT (ARGUMENT1 ARGUMENT2)
  (COND
    ((AND
       (NUMBERP ARGUMENT1)
       (NUMBERP ARGUMENT2))
     (* ARGUMENT1 ARGUMENT2))
    ((= ARGUMENT1 0) 0)
    ((= ARGUMENT2 0) 0)
    ((= ARGUMENT1 1) ARGUMENT2)
    ((= ARGUMENT2 1) ARGUMENT1)
    (T (LIST '* ARGUMENT1 ARGUMENT2))))
```

Een functie V-DELING die de vereenvoudiging van een deling uitvoert:

```
(DEFUN V-DELING (ARGUMENT1 ARGUMENT2)
  (COND
    ((= ARGUMENT1 0) 0)
    ((= ARGUMENT2 0) 'UNDEFINED)
    ((= ARGUMENT1 1) ARGUMENT2)
    ((= ARGUMENT2 1) ARGUMENT1)
    ((AND
       (NUMBERP ARGUMENT1)
       (NUMBERP ARGUMENT2))
     (/ ARGUMENT1 ARGUMENT2))
    (T (LIST '/ ARGUMENT1 ARGUMENT2))))
```

Het verslag van VEREENVOUDIG voor

```
(VEREENVOUDIG '(+ (- A 0) 0)):
```

```
1. ENTER VEREENVOUDIG
     UITDRUKKING = (+ (- A 0.) 0.)
  2. ENTER VEREENVOUDIG
       UITDRUKKING = (- A 0.)
    3. ENTER VEREENVOUDIG
         UITDRUKKING = A
    3. EXIT VEREENVOUDIG A
    3. ENTER VEREENVOUDIG
         UITDRUKKING = 0.
    3. EXIT VEREENVOUDIG 0.
  2. EXIT VEREENVOUDIG A
  2. ENTER VEREENVOUDIG
       UITDRUKKING = 0.
  2. EXIT VEREENVOUDIG 0.
1. EXIT VEREENVOUDIG A

Eindresultaat: A
```

En tenslotte het verslag voor

```
(VEREENVOUDIG '(* (/ 12 2) (* A 1)))
```

```
1. ENTER VEREENVOUDIG
     UITDRUKKING = (* (/ 12. 2.) (* A 1.))
  2. ENTER VEREENVOUDIG
       UITDRUKKING = (/ 12. 2.)
    3. ENTER VEREENVOUDIG 12.
         UITDRUKKING = 12
    3. EXIT VEREENVOUDIG 12.
    3. ENTER VEREENVOUDIG
         UITDRUKKING = 2.
    3. EXIT VEREENVOUDIG 2.
  2. EXIT VEREENVOUDIG 6.
  2. ENTER VEREENVOUDIG
       UITDRUKKING = (* A 1.)
    3. ENTER VEREENVOUDIG
         UITDRUKKING = A
    3. EXIT VEREENVOUDIG A
    3. ENTER VEREENVOUDIG
         UITDRUKKING = 1.
    3. EXIT VEREENVOUDIG 1.
  2. EXIT VEREENVOUDIG A
1. EXIT VEREENVOUDIG (* 6. A)

Eindresultaat: (* 6. A)
```

9.2.3. VAN INFIX NAAR PREFIX

Laten we eerst veronderstellen dat er geen preferentiële ordening is van de operatoren, m.a.w. bij twee opeenvolgende operatoren heeft de eerste altijd voorrang op de tweede. Door haakjes te gebruiken, kunnen we altijd een andere ordening aangeven.

Daarna zullen we veronderstellen dat haakjes mogen wegvallen voor opeenvolgende operatoren waarvan de tweede voorrang heeft. Bijvoorbeeld (a + b * c) valt hieronder omdat * voorrang heeft op +.

We beginnen met een programma dat van uitdrukkingen in infix-notatie naar uitdrukkingen in prefix-notatie gaat, gegeven dat de infix-uitdrukkingen niet dubbelzinnig zijn. De eenvoudigste oplossing volgt opnieuw uit het toepassen van de recursieve methode. Als we een infix-uitdrukking naar prefix willen omzetten, zijn er vooreerst twee gevallen:

1. De uitdrukking is een symbool of een getal, en dan hoeven we niets om te zetten; prefix- en infix-notatie zijn hier gelijk.
2. De uitdrukking is een lijst van operanden, afgewisseld met operatoren. In dat geval roepen we een hulpfunctie INF-TO-PRE-INTERNAL op, die de verdere omzetting zal doen.

INF-TO-PRE-INTERNAL heeft twee argumenten: UITDRUKKING en LINKER-OPERAND. UITDRUKKING is een stuk van een infix-uitdrukking dat begint met een operator. LINKER-OPERAND is een prefix-uitdrukking, die de linker operand vormt van de operator waarmee UITDRUKKING begint.

INF-TO-PRE-INTERNAL onderscheidt twee gevallen:

1. UITDRUKKING is gelijk aan de lege lijst. Er zijn dus geen operatoren meer, en het resultaat is gelijk aan de ene operand waaraan LINKER-OPERAND gebonden is.
2. UITDRUKKING is niet gelijk aan de lege lijst. In dat geval moet de operator aan het begin van UITDRUKKING met zijn twee operanden samengesmolten worden tot een prefix-uitdrukking, die bij de recursieve oproep de nieuwe linker operand van de volgende operator (twee plaatsen verder in UITDRUKKING) zal zijn.

   Die twee operanden vinden we als volgt: de linker operand staat reeds in prefix-notatie en is gebonden aan LINKER-OPERAND, de rechter operand is het naar prefix omgezette tweede element van UITDRUKKING.

In LISP ziet het er als volgt uit:

```
(DEFUN INF-TO-PRE (UITDRUKKING)
  (COND
    ((OR (SYMBOLP UITDRUKKING)
         (NUMBERP UITDRUKKING))
     UITDRUKKING)
    (T
      (INF-TO-PRE-INTERNAL (CDR UITDRUKKING)
                           (INF-TO-PRE (CAR UITDRUKKING))))))

(DEFUN INF-TO-PRE-INTERNAL (UITDRUKKING OPERAND)
  (COND
    ((NULL UITDRUKKING)
      OPERAND)
    (T
      (INF-TO-PRE-INTERNAL (CDDR UITDRUKKING)
                           (LIST
                             (CAR UITDRUKKING)
                             OPERAND
                             (INF-TO-PRE
                               (CADR UITDRUKKING)))))))
```

Tot nu toe klopt de omzetting alleen maar als bij twee opeenvolgende operatoren de eerste voorrang heeft op de tweede. Zo wordt (a * b + c) wel juist omgezet naar (+ (* a b) c), maar (a + b * c) naar (* (+ a b) c), wat fout is omdat * voorrang heeft op +. Nu verruimen we het bereik van dit programma door rekening te houden met de voorrang van de operatoren.

De eerste stap is de definitie van twee hulpfuncties. OPERATORP is een predikaat dat bepaalt of zijn argument een operator-symbool is of niet:

```
(DEFUN OPERATORP (OP)
  (MEMBER OP '(↑ // * - +)))
```

VOORRANG-P is een predikaat dat bepaalt of de eerste operator voorrang heeft op de tweede. Laten we veronderstellen dat de variabele OPERATOREN alle operatoren bevat, waarbij de operator met de meeste voorrang eerst komt:

```
(↑ / * - +)
```

VOORRANG-P kan dan als volgt worden gedefinieerd:

```
(DEFUN VOORRANG-P (OPERATOR1 OPERATOR2 OPERATOREN)
  (COND
   ((NULL OPERATOREN) NIL)
   ((EQ (CAR OPERATOREN) OPERATOR2) NIL)
   ((EQ (CAR OPERATOREN) OPERATOR1) T)
   (T (VOORRANG OPERATOR1 OPERATOR2
        (CDR OPERATOREN)))))
```

[Waarom testen we eerst of de tweede operator gelijk is aan het eerste element van de lijst?]

Het verschil met de vorige oplossing is dat we nu verder dan een operator diep in de uitdrukking moeten kijken, om iets over de volgorde van de bewerkingen te kunnen weten en zo de equivalente prefix-uitdrukking te vinden. Het reeds aangehaalde voorbeeld van (a + b * c) kan dit misschien verduidelijken.

Nadat we a, + en b gelezen hebben, kunnen we die twee operanden en de operator niet samensmelten tot (+ a b), omdat b eerst met c moet vermenigvuldigd worden, om dan pas het resultaat daarvan bij a op te tellen. Op een of andere manier moeten we dus a en + ergens 'onthouden' om ze in latere berekeningen terug te kunnen gebruiken.

Vermits de infix-uitdrukking willekeurig lang kan worden, en we dus niet op voorhand weten hoeveel operanden en operatoren we in de loop van de omzetting zullen moeten onthouden, voeren we twee stapelgeheugens ('stacks') in: één voor de operanden en één voor de operatoren. Telkens we een element van de infix-uitdrukking lezen stoppen we het op een van de twee stacks, waar het terug zal afgehaald worden op het ogenblik dat het in de prefix-uitdrukking kan ingepast worden.

Bij het schrijven van het programma gebruiken we terug twee functies: INF-TO-PRE, de top-level functie, en INF-TO-PRE-INTERNAL, de hulpfunctie die het eigenlijke omzetwerk doet. INF-TO-PRE is een nauwelijks veranderde versie van de gelijknamige functie uit de vorige oplossing. Alleen de argumenten waarmee INF-TO-PRE-INTERNAL opgeroepen wordt, zijn veranderd.

INF-TO-PRE-INTERNAL heeft nu drie argumenten: UITDRUKKING is nog steeds een lijst die het nog niet gelezen stuk van de infix-uitdrukking voorstelt; OPERAND-STACK is het stapelgeheugen dat de (reeds naar prefix-notatie omgezette) operanden bijhoudt, en OPERATOR-STACK is het stapelgeheugen waarop de operatoren bijgehouden worden.

Bij een oproep van INF-TO-PRE-INTERNAL kunnen we de volgende gevallen onderscheiden:

1. Als UITDRUKKING de lege lijst is, is er niets meer om te zetten, en hoeven we enkel nog de elementen van de twee stacks op de juiste manier te combineren. Dit gebeurt met de functie BOUW-UITDRUKKING.

2. Als de operator-stack meer dan één element bevat, en het bovenste element *geen* voorrang heeft op het tweede element, betekent dit dat de laatste twee operanden horen bij de op één na laatste operator. We kunnen die dus samensmelten tot een prefix-uitdrukking, die opnieuw op de top van de operand-stack gepushed wordt. De operator die we daarbij gebruikt hebben, halen we uit de operator-stack. Vervolgens roepen we INF-TO-PRE-INTERNAL recursief op met dezelfde UITDRUKKING, maar met de aangepaste stacks.

3. Als het eerste element van UITDRUKKING een operator is, pushen we het op OPERATOR-STACK, en roepen we INF-TO-PRE-INTERNAL recursief op met de rest van UITDRUKKING en de nieuwe stacks.

4. Anders is het eerste element van UITDRUKKING een operand, die we eerst naar prefix-notatie omzetten, en vervolgens op OPERAND-STACK pushen. Opnieuw roepen we INF-TO-PRE-INTERNAL recursief op, met de rest van UITDRUKKING en de nieuwe stacks.

In LISP ziet het er als volgt uit:

```
(DEFUN INF-TO-PRE (UITDRUKKING)
  (COND
   ((OR (SYMBOLP UITDRUKKING)
        (NUMBERP UITDRUKKING))
    UITDRUKKING)
   (T
    (INF-TO-PRE-INTERNAL UITDRUKKING NIL NIL))))
```

```
(DEFUN BOUW-UITDRUKKING (OPERANDS OPERATORS)
  (COND
    ((NULL OPERATORS)
      (CAR OPERANDS))
    (T
      (BOUW-UITDRUKKING
        (CONS
          (LIST (CAR OPERATORS)
                (CADR OPERANDS)
                (CAR OPERANDS))
          (CDDR OPERANDS))
        (CDR OPERATORS)))))

(DEFUN INF-TO-PRE-INTERNAL
          (UITDRUKKING OPERAND-STACK OPERATOR-STACK)
  (COND
    ((NULL UITDRUKKING)
      (BOUW-UITDRUKKING OPERAND-STACK OPERATOR-STACK))
    ((AND
       (>= (LENGTH OPERATOR-STACK) 2)
       (NOT (VOORRANG-P (CAR OPERATOR-STACK)
                        (CADR OPERATOR-STACK)
                        '(↑ // * - +))))
      (INF-TO-PRE-INTERNAL UITDRUKKING
                           (CONS
                             (LIST (CADR OPERATOR-STACK)
                                   (CADR OPERAND-STACK)
                                   (CAR OPERAND-STACK))
                             (CDDR OPERAND-STACK))
                           (CONS (CAR OPERATOR-STACK)
                                 (CDDR OPERATOR-STACK))))
    ((OPERATORP (CAR UITDRUKKING))
      (INF-TO-PRE-INTERNAL (CDR UITDRUKKING)
                           OPERAND-STACK
                           (CONS (CAR UITDRUKKING)
                                 OPERATOR-STACK)))
    (T
     (INF-TO-PRE-INTERNAL (CDR UITDRUKKING)
                          (CONS
                            (INF-TO-PRE (CAR UITDRUKKING))
                            OPERAND-STACK)
                          OPERATOR-STACK))))
```

Hier is een verslag van INF-TO-PRE-INTERNAL voor (INF-TO-PRE '(A + B * C - D)). Merk op hoe de OPERATOR-STACK en OPERAND-STACK veranderen. Merk ook op dat INF-TO-PRE-INTERNAL nog steeds een staartrecursie is.

```
1. ENTER INF-TO-PRE-INTERNAL
   (A + B * C - D) NIL NIL
 2. ENTER INF-TO-PRE-INTERNAL
    (+ B * C - D) (A) NIL
  3. ENTER INF-TO-PRE-INTERNAL
     (B * C - D) (A) (+)
   4. ENTER INF-TO-PRE-INTERNAL
      (* C - D) (B A) (+)
    5. ENTER INF-TO-PRE-INTERNAL
       (C - D) (B A) (* +)
     6. ENTER INF-TO-PRE-INTERNAL
        (- D) (C B A) (* +)
      7. ENTER INF-TO-PRE-INTERNAL
         (D) (C B A) (- * +)
       8. ENTER INF-TO-PRE-INTERNAL
          (D) ((* B C) A) (- +)
        9. ENTER INF-TO-PRE-INTERNAL
           NIL (D (* B C) A) (- +)
        9. EXIT INF-TO-PRE-INTERNAL
           (+ A (- (* B C) D))
       8. EXIT INF-TO-PRE-INTERNAL
          (+ A (- (* B C) D))
      7. EXIT INF-TO-PRE-INTERNAL
         (+ A (- (* B C) D))
     6. EXIT INF-TO-PRE-INTERNAL
        (+ A (- (* B C) D))
    5. EXIT INF-TO-PRE-INTERNAL
       (+ A (- (* B C) D))
   4. EXIT INF-TO-PRE-INTERNAL
      (+ A (- (* B C) D))
  3. EXIT INF-TO-PRE-INTERNAL
     (+ A (- (* B C) D))
 2. EXIT INF-TO-PRE-INTERNAL
    (+ A (- (* B C) D))
1. EXIT INF-TO-PRE-INTERNAL
   (+ A (- (* B C) D))

Eindresultaat: (+ A (- (* B C) D))
```

Er zijn nog enkele deelproblemen die niet behandeld zijn zoals operatoren die slechts één argument hebben (bijvoorbeeld NOT), en ongrammaticale uitdrukkingen (zoals (a a *)). Uitbreidingen van de definitie in deze zin zijn verdere oefeningen voor de geïnteresseerde lezer.

9.3. NOG ENKELE OPGAVEN

1. Veronderstel een lijst van steden. Tussen sommige steden is er een luchtvaartverbinding van een bepaalde duur. Schrijf een functie die bepaalt wat de snelste manier is om van de ene stad naar de andere te vliegen. Veronderstel 30 min overstaptijd.

2. Schrijf een functie die alle mogelijke combinaties van een lijst van elementen genereert.

3. Schrijf een functie die alle mogelijke permutaties van een lijst van elementen genereert.

4. LISP gebruikt een prefixnotatie, functies staan vooraan gevolgd door de argumenten. Bij een postfixnotatie staan functies achteraan. Bijvoorbeeld (5 10 +) is het equivalent in postfixnotatie van (+ 5 10). Schrijf een functie die formules van prefix- naar postfixnotatie omzet.

DEEL 3. IMPERATIEF PROGRAMMEREN

1. GETALLEN EN REKENKUNDIGE FUNCTIES
2. STRINGS
3. IMPERATIEF PROGRAMMEREN
4. PROG
5. GLOBALE VARIABELEN
6. DESTRUCTIEVE OPERATIES OVER LIJSTEN
7. COMPLEXE CONTROLESTRUCTUREN
8. SAMENVATTING
9. GEMENGDE OPGAVEN

Dit deel breidt het basisrepertorium verder uit met primitieve functies voor getallen en strings. Daarna bekijken we functies voor het construeren van een programma in imperatieve stijl: PROG, globale variabelen en destructieve operaties over lijsten.

1. GETALLEN EN REKENKUNDIGE FUNCTIES

1.1. DEFINITIES

1.1.1. TYPEN

Een LISP-systeem heeft diverse typen getallen. De belangrijkste zijn integers en floating-point numbers, dikwijls afgekort als flonums.

1. Een integer dient om gehele getallen voor te stellen. Het bereik hangt af van de implementatie. Een typisch voorbeeld is een bereik van -2^{31} tot $2^{31}-1$. Een bignum (afkorting van *big number*) dient om gehele getallen buiten dit bereik voor te stellen. Voor de gebruiker is er geen verschil tussen integer en bignum, LISP zet de getallen in de juiste vorm en de functies die voor integers werken, werken ook voor bignums.
2. Een flonum dient om reële getallen voor te stellen. 1.5, .707 en -3.14159 zijn voorbeelden. Het bereik hangt opnieuw af van de implementatie. Een typisch voorbeeld is een bereik van $\pm 2.9\times 10^{-37}$ tot $\pm 1.7\times 10^{38}$. Er is ook een exponentiële notatie van de vorm mEn, met m een flonum of integer en n een integer. Het getal is dan gelijk aan $n\times 10^{m}$. Bijvoorbeeld 1e-3 is gelijk aan 0.001. 1e3 is gelijk aan 1000.0

De meeste LISP-systemen, en dus ook COMMON LISP, hebben ook nog rationele getallen, voorgesteld als X/Y, en zelfs complexe getallen, voorgesteld als #C(x y). Deze worden hier niet verder behandeld. Overigens is het ook niet interessant om in de context van een inleidend werk de vele rekenkundige functies die normaal in een LISP-systeem te bespreken. We geven enkel voorbeelden van de belangrijkste functies.

1.1.2. REKENKUNDIGE FUNCTIES

De symbolen +, -, * en / representeren de operaties som, verschil, vermenigvuldiging en deling voor getallen. Het zijn generische functies die werken voor alle typen van getallen. Al deze rekenkundige functies kunnen een onbepaald aantal argumenten hebben. Bijvoorbeeld (+ 5 10 15) is gelijk aan 30. De

verschilfuncties trekken de som van de laatste argumenten af van het eerste argument. Dus (- 10 8 2) is gelijk aan 0.

(- X) is gelijk aan het negatieve equivalent van X. (1- X) trekt 1 af van een getal X en (1+ X) voegt 1 toe bij een getal X.

1.1.3. REKENKUNDIGE PREDIKATEN

De rekenkundige predikaten werken met gelijk welk type van getal.

(= X Y) levert T op als X gelijk is aan Y. Omzettingen worden gemaakt als een van de twee een integer is.

Merk op dat EQ niet werkt voor getallen omdat het alleen symbolen vergelijkt.

(> X_1 X_2 · · ·) vergelijkt de argumenten van links naar rechts en levert T op als X_n steeds groter is dan X_{n+1}, anders NIL.

(< X_1 X_2 · · ·) vergelijkt de argumenten van links naar rechts en levert T op als X_n steeds kleiner is dan X_{n+1}, anders NIL.

EVENP en ODDP gaan na of een getal even of oneven is.

ZEROP gaat na of een getal gelijk is aan 0.

1.1.4. EXPONENTIELE EN LOG FUNCTIES

(EXPT X Y) berekent X tot de Y-de macht. (SQRT X) berekent de vierkantswortel van een getal X. (LOG X) berekent de natuurlijke logaritme van X.

1.1.5. TRIGONOMETRISCHE FUNCTIES

(SIN X) berekent de sinus van X, uitgedrukt in radialen.

(SIND X) berekent de sinus van X, uitgedrukt in graden.

(COS X) berekent de cosinus van X, uitgedrukt in radialen.

(COSD X) berekent de cosinus van X, uitgedrukt in graden.

(ATAN Y X) berekent de hoek, uitgedrukt in radialen, waarvan de tangens gelijk is aan Y/X. ATAN levert altijd een positief getal op tussen 0 en 2π.

(ATAN2 Y X) berekent de hoek, uitgedrukt in radialen, waarvan de tangens gelijk is aan Y/X. ATAN2 levert altijd een getal op tussen π en $-\pi$.

1.2. OEFENINGEN

1. Wat is (/ 6.03e23 5.e20)?

1206.0.

2. Wat is (- (EXPT 5.5 3)
   (SQRT 256e2))?

6.375, immers (EXPT 5.5 3), dit wil zeggen 5.5 tot de derde macht, is gelijk aan 166.375. 256e2 is gelijk aan 25600.0 en de vierkantswortel hiervan is 160.0. Het verschil is 6.375.

3. Wat is (> 5 3 1)?

T want 5 is groter dan 3 en 3 groter dan 1.

4. Definieer de trigonometrische functie SECANS.

```
(DEFUN SECANS (X)
  (/ 1 (COS X)))
```

5. Definieer de functie + voor twee argumenten gebruik makend van 1+ en 1-, veronderstel dat het eerste argument een integer is.

Analyse: Er zijn drie gevallen. Ofwel is het eerste argument X gelijk aan 0, dan is het resultaat Y. Ofwel is X een negatief getal, dan moeten we recursief + toepassen waarbij we 1 bijtellen bij X en 1 aftrekken van het resultaat. Ofwel is X een positief getal, dan moeten we recursief + toepassen waarbij we 1 aftrekken van X en 1 bijtellen bij het resultaat.

```
(DEFUN + (X Y)
  (COND
    ((ZEROP X) Y)
    ((< X 0) (1- (+ (1+ X) Y)))
    (T (1+ (+ (1- X) Y)))))
```

6. Het statistisch gemiddelde van een reeks van n getallen is de som van deze getallen gedeeld door n. Schrijf een functie GEMIDDELDE die deze waarde berekent.

```
(DEFUN GEMIDDELDE (GETALLENREEKS)
  (/
    (SOM GETALLENREEKS)
    (LENGTH GETALLENREEKS)))
```

waarbij

```
(DEFUN SOM (GETALLENREEKS)
  (COND
    ((NULL GETALLENREEKS) 0)
    (T
      (+
        (CAR GETALLENREEKS)
        (SOM (CDR GETALLENREEKS))))))
```

7. Het aantal permutaties van een reeks van n objecten is gelijk aan

$$n\,! = n\,(n - 1)\,(n - 2) \ldots 3\ 2\ 1$$

Schrijf een functie FACTORIAL die deze waarde berekent.

```
(DEFUN FACTORIAL (N)
  (IF (ZEROP 0) 1 (* N (FACTORIAL (1- N)))))
```

8. Het aantal manieren waarop r objecten kunnen geselecteerd worden uit n objecten is gelijk aan $\binom{n}{r} = \frac{n!}{(n-r)!\ r!}$

Definieer de functie COMBINATIES die deze waarde berekent.

```
(DEFUN COMBINATIES (N R)
  (/
    (FACTORIAL N)
    (*
      (FACTORIAL (- N R))
      (FACTORIAL R))))
```

9. Euclides heeft een algoritme bedacht om de grootste gemene deler van twee getallen a en b te berekenen. Het algoritme is gebaseerd op het feit dat ggd(a,b) = ggd(b,r), waarbij r de rest is van a/b, en dat ggd(a,0) = a. Als we dus opeenvolgende reducties uitvoeren tot b gelijk is aan 0 kunnen we gemakkelijk de grootste gemene deler berekenen. Implementeer dit algoritme in LISP. Veronderstel een functie REMAINDER.

```
(DEFUN GGD (A B)
  (IF (ZEROP B) A (GGD B (REMAINDER A B))))
```

10. Newton heeft een algoritme bedacht voor het berekenen van de vierkantswortel van een getal x. Zijn idee is om een mogelijke oplossing te raden en die dan te verbeteren door het gemiddelde te nemen van de benadering en x gedeeld door de benadering, tot ze voldoet. Implementeer dit algoritme voor een precisie van 0.001.

We beginnen met als eerste benadering het getal gedeeld door 2.

```
(DEFUN VIERKANTSWORTEL (GETAL)
  (VIERKANTSWORTEL2 (/ GETAL 2) GETAL))
```

VIERKANTSWORTEL2 verbetert recursief de benadering tot ze goed genoeg is.

```
(DEFUN VIERKANTSWORTEL2 (BENADERING GETAL)
  (COND
   ((GOED-GENOEG BENADERING GETAL) BENADERING)
  (T
   (VIERKANTSWORTEL2
    (BETERE BENADERING GETAL) GETAL))))
```

Een benadering is goed genoeg als het verschil tussen de benadering tot de tweede macht en het getal zelf kleiner is dan 0.001.

```
(DEFUN GOED-GENOEG (BENADERING GETAL)
  (< (ABSOLUTE-WAARDE
      (- (EXPT BENADERING 2) GETAL))
     0.001))
```

BETERE berekent een betere benadering:

```
(DEFUN BETERE (BENADERING GETAL)
  (GEMIDDELDE
    BENADERING
    (/ GETAL BENADERING)))
```

11. De wortels van een vergelijking

$$ax^2 + bx + c = 0$$

zijn gelijk aan

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Schrijf een functie WORTELS die de twee wortels van een vergelijking berekent, gegeven A, B en C. Veronderstel dat A, B en C verschillend zijn van 0.

De functie berekent eerst $\sqrt{b^2 - 4ac}$ als deelresultaat om dubbele evaluatie te vermijden.

```
(DEFUN WORTELS (A B C)
  (LET
    ((DEELRESULTAAT
       (SQRT
         (-
           (EXPT B 2)
           (* 4 A C)))))
    (LIST
      (/
        (+ (- B) DEELRESULTAAT)
        (* 2 A))
      (/
        (- (- B) DEELRESULTAAT)
        (* 2 A)))))
```

12. Schrijf een functie die het afgewogen gemiddelde berekent van een reeks getallen, gebruik makend van de formule

$$\frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$$

De functie heeft twee argumenten: een lijst van de factors en een lijst van de waarden.

```
(DEFUN WEIGHTED-MEAN (FACTORS WAARDEN)
 (/
  (APPLY '+
         (MAPCAR
           '(LAMBDA (FACTOR WAARDE)
              ( * FACTOR WAARDE))
           FACTORS WAARDEN))
  (APPLY '+ FACTORS)))
```

2. STRINGS

2.1. DEFINITIES

LISP heeft ook primitieve functies om te werken met strings. Strings worden gedefinieerd met "". Bijvoorbeeld "abc" is een string. In zekere zin functioneren symbolen als strings. Een symbool is echter meer dan een string omdat het ook eigenschappen kan hebben zoals een binding of een functiedefinitie. De string die correspondeert met een symbool heet de PNAME (print name) van dit symbool. Stringfuncties werken ook op symbolen, door ze eerst om te zetten naar hun pnames.

(SYMBOL-NAME X) met X een symbool is gelijk aan de PNAME van X. Bijvoorbeeld (SYMBOL-NAME 'ABC) is gelijk aan "ABC".

(INTERN X) met X een string is het omgekeerde van SYMBOL-NAME. Evaluatie is gelijk aan een symbool waarvan de pname gelijk is aan de string X. Bijvoorbeeld (INTERN "ABC") is gelijk aan ABC.

(STRING-LESSP X Y) met X en Y twee strings (of symbolen) is gelijk aan T als X alfabetisch voor Y komt.

De functie GENSYM dient om nieuwe symbolen te maken. Bijvoorbeeld (GENSYM) is gelijk aan G0001, de volgende oproep van GENSYM levert G0002, enzovoort. GENSYM met een argument (een string) gebruikt dat argument als prefix voor het nieuw gegenereerde symbool. Bijvoorbeeld (GENSYM "SYM#") is SYM#0032.

2.2. OEFENINGEN

1. Definieer een functie NIEUW-SYMBOOL die een nieuw symbool maakt van twee symbolen. Bijvoorbeeld (NIEUW-SYMBOOL 'ABC 'DEF) is gelijk aan ABCDEF. Gebruik de functie STRING-APPEND om twee strings te concateneren.

```
(DEFUN NIEUW-SYMBOOL (X Y)
  (INTERN
    (STRING-APPEND (SYMBOL-NAME X)
                   (SYMBOL-NAME Y))))
```

2. Definieer een functie (SUBSYMBOOL X Y) die nagaat of de pname van X een substring is van de pname van Y. Gebruik de functie STRING-SEARCH die iets anders dan NIL terguggeeft als het eerste argument een substring van het tweede argument is.

```
(DEFUN SUBSYMBOOL (X Y)
   (STRING-SEARCH (SYMBOL-NAME X) (SYMBOL-NAME Y)))
```

3. Definieer een functie ALFABETISCH-EERSTE die gegeven een lijst symbolen het symbool oplevert dat alfabetisch het eerste komt. Bijvoorbeeld (ALFABETISCH-EERSTE '(ADAM EVA ABRAHAM)) is gelijk aan ABRAHAM.

   We schrijven een functie EERSTE2 die via staartrecursie het bedoelde symbool vindt.

```
(DEFUN ALFABETISCH-EERSTE (SYMBOLEN)
  (EERSTE2 (CDR SYMBOLEN) (CAR SYMBOLEN)))
```

   EERSTE2 vertrekt van een hypothese en wijzigt deze hypothese als het een nieuw symbool ontmoet dat vroeger komt.

```
(DEFUN EERSTE2 (SYMBOLEN EERSTE)
 (COND
  ((NULL SYMBOLEN) EERSTE)
  ((STRING-LESSP (CAR SYMBOLEN) EERSTE)
   (EERSTE2 (CDR SYMBOLEN) (CAR SYMBOLEN)))
  (T (EERSTE2 (CDR SYMBOLEN) EERSTE))))
```

3. IMPERATIEF PROGRAMMEREN

LISP is een applicatieve taal. De basisoperatie in het evaluatieproces is het toepassen van een functie op een reeks argumenten. Dit levert een waarde op die dan weer een rol speelt in verdere evaluatieprocessen of als eindresultaat aan de gebruiker wordt meegedeeld.

Imperatieve talen (zoals FORTRAN en PASCAL) zijn talen die gebaseerd zijn op bevelen. Een bevel wordt niet uitgevoerd omwille van de waarde die dit bevel oplevert (het is vreemd om over bevelen te denken in termen van resulterende waarden) maar omwille van het *neveneffect*, d.w.z. de wijziging die het aanbrengt in het geheugen of in het controleverloop. Imperatief programmeren bestaat uit het organiseren van bevelen zodat het juiste neveneffect bereikt wordt op het juiste moment. Alhoewel LISP een applicatieve taal is, kan men ook imperatief programmeren in LISP.

Er zijn een aantal ernstige problemen verbonden met een imperatieve stijl van programmeren.

1. *Modulariteit is niet langer gegarandeerd.* Modulariteit wil zeggen dat men twee stukken apart construeert en daarna samenvoegt zonder dat er onvoorziene dingen gebeuren of er iets veranderd moet worden aan één van de stukken. De LISP-functies uit vorige delen garanderen modulaire programma's, imperatieve functies doen dit niet. Het gebrek aan modulariteit in imperatieve talen is één van de belangrijkste bronnen van fouten, vooral in grote systemen die door teams van programmeurs ontwikkeld worden.
2. *Het is moeilijker om interactief te programmeren.* Applicatieve talen zijn van nature interactief. Het evaluatieproces kan immers op gelijk welk moment beginnen of stoppen. Als een fout ontstaat is het mogelijk om de bindingen of een functie te wijzigen en voort te gaan met evaluatie. Dit levert enorme tijdswinst op tijdens het programmeren.
3. *Imperatieve talen zijn van nature niet-interactief* (alhoewel er ook semi-interactieve imperatieve talen zijn). De stand van zaken kan niet meer

eenvoudig beschreven worden door een formule en een lijst van bindingen. Ook is het niet gemakkelijk om stappen terug te zetten omdat dan alle cellen die werden veranderd door neveneffecten opgespoord en hersteld moeten worden.

4. *Er zijn problemen bij het opbouwen van een semantiek voor imperatieve talen.* Applicatieve talen hebben een logische fundering, vandaar dat de axiomatische semantiek van deze talen relatief gemakkelijk te formaliseren is en dat het ook relatief gemakkelijk is om formeel na te denken over programma's. Het is bijvoorbeeld niet moeilijk om te bewijzen dat (APPEND X (APPEND Y Z)) gelijk is aan (APPEND (APPEND X Y) Z)[1] Anderzijds is het moeilijk gebleken om een praktische bewijstheorie voor imperatieve talen te ontwikkelen. De alledaagse programmeur ondervindt deze moeilijkheden indirect. Alhoewel hij niet probeert de correctheid van zijn programma's te bewijzen, vormt de onoverzichtelijke semantiek een hinderpaal bij het nadenken over imperatieve programma's.

5. *Efficiëntie.* Het is wel zo dat imperatieve talen efficiënter zijn op de huidige generatie computers. Daarom zijn er LISP compilers die een applicatief programma omzetten in een imperatief programma dat dan wel efficiënt loopt, incluis voor numerieke toepassingen. Imperatieve talen zijn in principe efficiënter omdat de meeste commerciële computers von Neumann machines zijn.

Een von Neumann machine bevat drie componenten: een centrale verwerkingseenheid die bevelen kan uitvoeren, een geheugen dat gegevens en tussentijdse resultaten kan opslaan, en een kanaal voor communicatie tussen de twee. De centrale verwerkingseenheid voert bevelen uit die de toestand van het geheugen veranderen. Dit geeft aanleiding tot programma's die bestaan uit opeenvolgende wijzigingen van een geheugentoestand en programmeertalen die toelaten zo'n programma's gemakkelijk te schrijven. Imperatieve talen zijn directe reflecties van deze architectuur en zijn dus zeer efficiënt op de Von Neumann computer.

Als we echter een andere computer architectuur gebruiken zijn applicatieve talen niet langer meer inefficiënt, integendeel. Het is bijvoorbeeld mogelijk om formules parallel te evalueren zodat de evaluatie sneller verloopt. Op dit moment wordt in verschillende laboratoria gewerkt aan nieuwe computers (reductiemachines,

---

[1] Cfr. de voorbeelden in Boyer en Moore (1975)

boommachines, data flow computers) die bijzonder geschikt zijn voor het interpreteren van applicatieve talen. Sommige hebben een applicatieve taal als machinetaal! In de nabije toekomst kan er op dit vlak een zeer interessante ontwikkeling worden verwacht.

Dit wil natuurlijk niet zeggen dat imperatieve talen van geen nut zouden zijn. Maar het zijn wel goede redenen om applicatieve talen verder te exploreren en om imperatief programmeren in LISP zoveel mogelijk te vermijden.

4. PROG

4.1. DEFINITIES

Hier is een voorbeeld van een imperatief programma voor het berekenen van de lengte van een lijst:

1. Zet een variabele L gelijk aan de gegeven lijst. Zet een andere variabele N gelijk aan 0.
2. Als L de lege lijst is, dan is het eindresultaat gelijk aan N.
3. Anders voeg 1 toe bij N.
4. Zet L gelijk aan de rest van L.
5. Begin opnieuw vanaf lijn 2.

Om programma's in deze stijl te schrijven moeten we instructies kunnen ordenen, iets gelijk kunnen zetten aan een variabele, kunnen terugkeren naar een vroegere instructie en een iteratie kunnen beëindigen met een bepaalde waarde. De PROG (afkorting van 'program') is de basisfunctie voor het schrijven van een stuk programma in imperatieve stijl en bevat constructies voor deze activiteiten.

Een PROG begint met een aantal variabelen (de zogenaamde PROG-variabelen). Deze variabelen zijn locaal voor de PROG en worden gelijk gezet aan NIL. De manier om een variabele gelijk te zetten aan een bepaalde waarde gebeurt met een functie SETQ die als neveneffect de waarde van een variabele verandert. De formules in een PROG zijn geordend en worden één voor één uitgevoerd. Tussen de formules mogen symbolen voorkomen die als *labels* dienen. De functie GO met een label als argument zorgt ervoor dat het uitvoeren van instructies herbegint met de formule vlak na het label. RETURN tenslotte is een functie om de iteratie te beëindigen. RETURN heeft een argument voor het eindresultaat.

Hier is een voorbeeld van het LENGTH programma:

```
(PROG (L N)
  (SETQ L INPUT-LIJST)
  (SETQ N 0)
 BEGIN
  (COND
    ((NULL L) (RETURN N)))
  (SETQ N (+ 1 N))
  (SETQ L (CDR L))
  (GO BEGIN))
```

L en N zijn PROG-variabelen. In de eerste twee formules worden L en N geïnitialiseerd met de INPUT-LIJST en 0. BEGIN is een label. De volgende instructie in de PROG is een voorwaardelijke uitdrukking. Als L gelijk is aan de lege lijst, stopt de iteratie door RETURN. N is dan gelijk aan de waarde. Anders wordt N gelijk gezet aan N + 1, L wordt gelijk gezet aan de rest van L en evaluatie begint opnieuw met de formule vlak na BEGIN, d.w.z. de voorwaardelijke uitdrukking.

Lezers die vertrouwd zijn met imperatieve talen zien onmiddellijk dat dit programma niet verschilt van soortgelijke programma's in imperatieve talen zoals FORTRAN, behalve dan de syntaxis uiteraard.

CONTROLE STRUCTUREN

Een elegantere manier om een iteratie op te schrijven is via een functie DO. DO heeft drie componenten. De eerste component bevat variabelen zoals PROG, een manier om ze te initialiseren en een step-functie die zegt wat de variabele wordt in de volgende iteratie. De tweede component is een predikaat dat als eindconditie dienst doet. Als het predikaat waar is wordt het eindresultaat berekend door evaluatie van de eindformule. De derde component van de DO-loop is een reeks formules die uitgevoerd worden bij elke iteratie:

```
(DO
  ((variabele1 initialisatie1 step1)
   (variabele2 initialisatie2 step2)
   ... )
  (predikaat eindformule)
  formule ... formule)
```

DO realiseert (zoals LET) alle bindingen in parallel. Indien men bij een binding reeds de waarde van een andere variabele nodig heeft, moeten de bindingen serieel gerealiseerd worden. Daarvoor bestaat er DO* (naar analogie met LET*).

Het programma voor LENGTH ziet eruit als volgt:

```
(DO
  ((L INPUT-LIJST (CDR L))
   (N 0 (+ 1 N)))
  ((NULL L) N))
```

Er zijn geen formules in deze DO. L wordt geïnitialiseerd met de input-lijst en N met 0. Bij elke iteratie wordt L gelijk aan de rest van L en N wordt met 1 vermeerderd. Als de lijst leeg is stopt de iteratie. De lengte van de lijst is gelijk aan N.

Een functie zoals DO maakt het controleverloop van het programma duidelijk. Het is zeer gemakkelijk om in LISP functies zoals DO, WHILE, FOR en andere typische controlestructuren te introduceren. De programmeur kan daarom gemakkelijk structuur aanbrengen in de beschrijving van het controleverloop van zijn programma.

4.2. OEFENINGEN

1. Definieer de functie MEMBER via een PROG.

Algoritme:

1. Zet de lijst gelijk aan L.
2. Als L de lege lijst is, is het resultaat NIL.
3. Als het eerste element van L gelijk is aan het element is het resultaat T.
4. Zet L gelijk aan de rest van L en begin opnieuw vanaf 2.

In LISP:

```
(DEFUN MEMBER (ELEMENT LIJST)
  (PROG (L)
    (SETQ L LIJST)
   BEGIN
    (COND
      ((NULL L) (RETURN NIL))
      ((EQ (CAR L) ELEMENT) (RETURN T)))
    (SETQ L (CDR L))
    (GO BEGIN)))
```

2. Definieer de functie REVERSE met een DO.

L gaat iteratief door de lijst. M houdt telkens het eerste element bij. Als L leeg is bevat M het resultaat.

```
(DEFUN REVERSE (LIJST)
  (DO*
    ((L LIJST (CDR L))
     (M NIL (CONS (CAR L) M)))
    ((NULL L) M)))
```

3. Definieer een functie VOEG-TOE die de elementen van een lijst toevoegt aan een andere lijst. Gebruik DO.

```
(DEFUN VOEG-TOE (LIJST1 LIJST2)
  (DO*
   ((L LIJST1 (CDR L))
    (M LIJST2 (CONS (CAR L) M)))
   ((NULL L) M)))
```

4. Wat is volgens deze definitie (VOEG-TOE '(A B) '(A B C D))?

(B A A B C D), en niet (A B A B C D). Hier zijn de bindingen van L en M in de evaluatie:

```
L = (A B)
M = (A B C D)

L = (B)
M = (A A B C D)

L = ()
M = (B A B C D)
```

5. Definieer een functie SPLITS-GETALLEN die gegeven een lijst met getallen en symbolen een paar teruggeeft waarbij het eerste element de lijst van de getallen en het tweede element de lijst van symbolen is, beide in dezelfde volgorde als in de oorspronkelijke lijst. Bijvoorbeeld (SPLITS-GETALLEN '(A 1 2)) ⇒ ((1 2) (A B)). Gebruik een PROG.

```
(DEFUN SPLITS-GETALLEN (LIJST)
  (PROG (L GETALLEN SYMBOLEN)
    (SETQ L LIJST)
   BEGIN
    (IF (NUMBERP (CAR L))
        (SETQ GETALLEN (CONS (CAR L) GETALLEN))
        (SETQ SYMBOLEN (CONS (CAR L) SYMBOLEN)))
    (SETQ L (CDR L))
    (IF (NULL L)
        (RETURN (LIST (RETURN GETALLEN)
                      (RETURN SYMBOLEN)))
        (GO BEGIN))))
```

6. Schrijf een PROG-functie VERSCHILLENDE-ELEMENTEN die nagaat hoeveel verschillende elementen er zijn in een lijst.

Dit is een imperatieve transcriptie van de definitie uit deel 2.

```
(DEFUN VERSCHILLENDE-ELEMENTEN (LIJST)
  (PROG (N L GEHEUGEN)
    (SETQ N 0)
    (SETQ L LIJST)
    (SETQ GEHEUGEN NIL)
   BEGIN
    (COND
      ((NULL L) (RETURN N))
      ((NOT (MEMBER (CAR L) GEHEUGEN))
       (SETQ N (1+ N))
       (SETQ GEHEUGEN (CONS (CAR L) GEHEUGEN))))
    (SETQ L (CDR L))
    (GO BEGIN)))
```

7. Construeer een DO versie van dezelfde functie.

```
(DEFUN VERSCHILLENDE-ELEMENTEN (LIJST)
  (DO*
    ((L LIJST (CDR L))
     (N 0 (COND
            ((NOT (MEMBER (CAR L) GEHEUGEN))
             (+ N 1))
            (T N)))
     (GEHEUGEN
       NIL
       (COND
         ((NOT (MEMBER (CAR L) GEHEUGEN))
           (CONS (CAR L) GEHEUGEN))
         (T GEHEUGEN))))
    ((NULL L) N)))
```

8. Schrijf een functie die uitdrukkingen leest, evalueert en het resultaat op het scherm brengt zoals een normaal LISP-systeem, behalve dat er >> verschijnt zodat de gebruiker weet wanneer er een formule moet worden ingetikt.

```
(DEFUN NIEUW-TOPLEVEL ()
  (PROG ()
    BEGIN
     (PRINT '>>)
     (PRINT (EVAL (READ)))
     (GO BEGIN)))
```

5. GLOBALE VARIABELEN

5.1. DEFINITIES

We hebben in deel 2 een onderscheid gemaakt tussen een LISP-systeem met een dynamisch en een lexicaal bereik. De bindingen voor een lexicaal bereik zijn enkel geldig in de functie waarin ze werden gebonden. De bindingen voor een dynamisch bereik blijven gelden voor de evaluatie van deeluitdrukkingen maar verdwijnen weer als de definitie is geëvalueerd. Het is mogelijk om nog een stap verder te gaan en zogenaamde *globale* variabelen te gebruiken die blijven gelden zelfs als de definitie is geëvalueerd.

Er zijn twee functies om globale variabelen in te voeren:

1. (DEFVAR naam [initiële-waarde [documentatie]]) introduceert een variabele met de gegeven naam en geeft ook optioneel een initiële waarde en optioneel een string met documentatie. Bijvoorbeeld:

   (DEFVAR *CURSOR-POSITION* '(1 1) "Positie van de cursor")

   introduceert de variabele *CURSOR-POSITION* met initiële waarde 1.

2. (DEFCONSTANT naam initiële-waarde [documentatie]) introduceert een variabele en geeft ook een initiële waarde (verplicht). Opnieuw is er de mogelijkheid om documentatie met de variabele te associëren. De variabele kan later wel niet meer van binding veranderen. Bijvoorbeeld:

   (DEFCONSTANT *PI* 3.45)

   introduceert de constante *PI* met binding 3.45.

Het is gebruikelijk in LISP om ** rond de namen van globale variabelen te plaatsen, maar dit is slechts een conventie.

De functie SETQ dient om de waarde van een globale variabele te veranderen. SETQ komt overeen met het "assignment-statement" in de klassieke programmeertalen.

De functie SETQ heeft twee argumenten: de naam van een variabele en een formule. SETQ evalueert de formule maar niet de naam van de variabele en zet het resultaat gelijk aan de variabele. De waarde van SETQ is gelijk aan het resultaat.

Bijvoorbeeld:

(SETQ *CURSOR-POSITION* (MOVE-CURSOR-TO-BOTTOM))

zal de binding van de variabele *CURSOR-POSITION* veranderen naar het resultaat van de evaluatie van de functie MOVE-CURSOR-TO-BOTTOM.

SETQ gaat automatisch veronderstellen dat het eerste argument een globale variabele is, zelfs als die niet door DEFVAR was gedeclareerd. Een variant van SETQ is SET. SET evalueert wel de naam van de variabele. De Q in SETQ komt van QUOTE.

SETQ wordt (nogal verwarrend) ook binnen PROG gebruikt maar dan is de binding enkel geldig binnen het bereik van de PROG als de variabele een PROG-variabele is. Bijvoorbeeld de volgende definitie van MEMBER

```
(DEFUN MEMBER (ELEMENT LIJST)
  (PROG ()
    BEGIN
     (COND
       ((NULL LIJST) (RETURN NIL))
       ((EQ (CAR LIJST) ELEMENT) (RETURN T)))
     (SETQ LIJST (CDR LIJST))
     (GO BEGIN)))
```

geeft wel het vereiste resultaat maar verandert LIJST in NIL. Bijgevolg zal

```
(IF (MEMBER 'A LIJST) LIJST NIL)
```

altijd NIL opleveren zelfs als A geen element is van de lijst. Dit is een goed voorbeeld hoe globale variabelen leiden tot ongewenste neveneffecten. Om dit te vermijden moet LIJST binnen het bereik van de PROG:

```
(DEFUN MEMBER (ELEMENT LIJST)
  (PROG (L)
    (SETQ L LIJST)
   BEGIN
    (COND
      ((NULL L) (RETURN NIL))
      ((EQ (CAR L) ELEMENT) (RETURN T)))
    (SETQ L (CDR L))
    (GO BEGIN)))
```

In LISP worden globale variabelen beschouwd als een symptoom van slechte programmeerstijl. Het is best om globale variabelen zo weinig mogelijk te gebruiken, enkel maar om informatie op te slaan die echt globaal is.

5.2. OEFENINGEN

1. Wat is het resultaat van (FOO 5) gegeven

```
(DEFVAR X)

(DEFUN FOO (X)
  (SETQ X 10)
  (BAR 5)
  X)
```

```
(DEFUN BAR (Y)
  (SETQ X 7))?
```

7, in het begin is X gebonden aan 5. In FOO wordt X gelijk gezet aan 10. In BAR wordt X opnieuw veranderd in 7 en X is het eind-resultaat van FOO.

2. Gegeven

```
(DEFVAR Y)

(DEFUN F1 (X)
  (SETQ Y 0)
  (F2 X)
  (F3 (+ 1 X))
  Y)

(DEFUN F2 (M) (SETQ Y (+ Y M)))

(DEFUN F3 (N) (SETQ Y (+ Y N)))
```

wat is het resultaat van (F1 4)?

---

9. Want (F2 4) zet Y gelijk aan 4. F3 voegt 5 toe aan Y.

3. Wat is X na evaluatie van

```
(DEFVAR X 5)
(DO () ((< X 0)) (SETQ X (1- X)))
```

-1, immers X wordt gelijk gezet aan X min 1 tot X kleiner is dan 0.

6. DESTRUCTIEVE OPERATIES OVER LIJSTEN

6.1. DEFINITIES

De operaties voor de constructie van lijsten die we tot hiertoe hebben bestudeerd, zijn *niet-destructief*, in de zin dat de argumenten van deze functies niet worden veranderd. Een functie zoals APPEND neemt elementen uit het eerste argument en voegt ze één voor één toe aan de tweede lijst. Omdat de argumentlijsten zelf niet veranderen, blijven andere programma's die ook verwijzen naar die lijsten werken. Op die manier blijft modulariteit gelden.

Bijvoorbeeld,

```
(SETQ X '(A B C))
(APPEND X X)
(PRINT X)
```

brengt (A B C) op het scherm en niet (A B C A B C) omdat X zelf niet verandert in APPEND.

LISP heeft ook *destructieve* operaties die de lijsten wel veranderen.

1. RPLACA verandert het eerste element van een lijst. Bijvoorbeeld (RPLACA '(A B C) 'D) is gelijk aan (D B C).

2. RPLACD verandert de cdr van een lijst. Bijvoorbeeld (RPLACD '(A B C) '(D E)) is gelijk aan (A D E).

3. NCONC voegt twee lijsten bij elkaar. Bijvoorbeeld (NCONC '(A B) '(D E)) is gelijk aan (A B D E).

In tegenstelling tot functies zoals APPEND veranderen deze functies *wel* de argumentlijsten. Bijvoorbeeld:

```
(LET
  ((L '(A B C)))
  (NCONC L L)
  L)
```

is gelijk aan (A B C A B C ...) (een circulaire lijst) omdat L impliciet verandert, terwijl

```
(LET
  ((L '(A B C)))
  (APPEND L L)
  L)
```

gelijk is aan (A B C) omdat L niet verandert.

De destructieve operaties zijn uiterst gevaarlijk en leiden snel tot fouten. Ze zijn soms wel nodig omwille van de efficiëntie.

6.2. OEFENINGEN

1. Wat is (RPLACA '(A B C) '(A B C))?

((A B C) B C)

2. Schrijf een niet-destructieve functie REPLACE-CAR die hetzelfde doet als RPLACA.

```
(DEFUN REPLACE-CAR (X Y)
  (CONS Y (CDR X)))
```

3. Wat is (RPLACD '(A B C) NIL)?

(A).

4. Wat is de evaluatie van

```
(LET
  ((X (NCONC '(A B) '(C D))))
  (RPLACA X 'D)
  X)
```

(D B C D), omdat (NCONC '(A B) '(C D)) gelijk is aan (A B C D) en (RPLACA X 'D) gelijk is aan (D B C D).

5. Definieer een functie MAAK-CIRCULAIRE-LIJST die een circulaire lijst maakt van een gegeven lijst.

```
(DEFUN MAAK-CIRCULAIRE-LIJST (L)
  (PROG (LIJST)
    (SETQ LIJST L)
    (RETURN (NCONC L L))))
```

6. Definieer een functie die een element X destructief vervangt door een element Y in een lijst L. Gebruik de DO-constructie.

De DO-variabele M loopt door de lijst L. Iteratie stopt als M de lege lijst is en dan is het resultaat gelijk aan L. Als het eerste element van M gelijk is aan X wordt dit element destructief vervangen door Y:

```
(DEFUN DESTRUCTIEVE-SUBST (X Y L)
  (DO*
    ((M L (CDR M)))
    ((NULL M) L)
    (COND
      ((EQ (CAR M) X)
        (RPLACA M Y)))))
```

7. Definieer een functie (SPLICE E P L) die op een destructieve wijze een element E op een positie P zet in een lijst L. L bevat minstens één element. Als P groter is dan de lengte van de lijst komt het element achteraan. Bijvoorbeeld, gegeven (SETQ M '(A)), dan is na (SPLICE 'D 1 M) M gelijk aan (D A), na (SPLICE 'E 2 M) is M gelijk aan (D E A) en na (SPLICE 'F 10 M) is M gelijk aan (D E A F).

Analyse: Als het element op de eerste plaats moet komen, d.w.z. als POSITIE gelijk is aan 1, dan moeten we eerst de CDR van de eerste cel in de lijst vervangen door de bestaande lijst, en daarna de CAR van de eerste cel vervangen door het nieuwe element. Als het element op de tweede plaats moet komen dan moeten we de rest van de lijst vervangen door de rest met het element eraan toegevoegd. Als de lijst slechts één element heeft voegen we het element toe aan de lijst, immers POSITIE is groter dan de lengte van de lijst. Anders passen we de functie recursief toe op de rest van de lijst en de positie min 1.

```
(DEFUN SPLICE (ELEMENT POSITIE LIJST)
  (COND
    ((= POSITIE 1)
      (RPLACD
        LIJST
        (CONS (CAR LIJST) (CDR LIJST)))
      (RPLACA LIJST ELEMENT))
    ((= POSITIE 2)
      (RPLACD LIJST (CONS ELEMENT (CDR LIJST))))
    ((NULL (CDR LIJST))
      (RPLACD LIJST (LIST ELEMENT)))
    (T
      (SPLICE
        ELEMENT
        (1- POSITIE)
        (CDR LIJST)))))
```

7. COMPLEXE CONTROLESTRUCTUREN

7.1. DEFINITIES

De belangrijkste controlestructuren van LISP, zoals de recursie of de imperatieve DO-constructies, zijn sterk gestructureerd. Het is niet mogelijk om vanuit een bepaalde eenheid (bijvoorbeeld een functie) naar een willekeurig punt te springen in een andere eenheid. De PROG maakt het al mogelijk om het punt van controle binnen een reeks van formules te doen verspringen via de GO. Maar het verschuiven van controle blijft binnen de PROG zelf die alleen kan worden verlaten met een RETURN.

In sommige toepassingen (vooral systeemprogrammatuur, proces-controle, real-time programmatie, enz.) is het nochtans noodzakelijk om meer complexe en ongestructureerde controlestructuren te gebruiken. Het kan bijvoorbeeld nodig zijn om in error-situaties ergens vanuit een programma naar een punt te springen dat buiten het programma werd gedefinieerd. Men noemt dit dynamische non-locale exits. Zij worden in LISP gerealiseerd met de functies CATCH and THROW.

1. (CATCH indicator lijst-van-formules) definieert een controlepunt dat geassocieerd wordt met de indicator. Daarna wordt de lijst van formules geëvalueerd alsof het ging om een gewone PROG tenzij een 'THROW' gebeurt.

2. (THROW indicator formule) brengt een exit tot stand. Controle verspringt naar het controlepunt aangegeven door de indicator. De formule beschrijft het resultaat dat terugkomt als er een exit is naar dit controlepunt.

We zouden bijvoorbeeld een top-level kunnen maken waar naartoe kan gesprongen worden vanuit errors of controlekarakters. Veronderstel het volgende standaard top-level:

```
(DEFUN TOP-LEVEL ()
  (PROG ()
    BEGIN
     (PRINT (EVAL (READ)))
     (GO BEGIN)))
```

We kunnen dit inpakken als volgt:

```
(DEFUN PROTECTED-TOP-LEVEL ()
  (PROG ()
    PROTECTED-BEGIN
     (CATCH 'TOP-LEVEL (TOP-LEVEL))
     (GO PROTECTED-BEGIN)))
```

Functies die willen terugspringen naar het PROTECTED-BEGIN kunnen dit doen met THROW:

```
(DEFUN CONTROL-G ()
  (THROW 'TOP-LEVEL 'CONTROL-G))
```

of

```
(DEFUN ERROR (ERROR-MESSAGE)
  ;; .... print hier error-message ...
  (THROW 'TOP-LEVEL 'ERROR))
```

De eerste functie CONTROL-G wordt opgeroepen als de gebruiker de Controle toets van het toetsenbord induwt gevolgd door het karakter G. Op dat moment springt controle onmiddellijk naar het toplevel en laat alle tussentijdse berekeningen voor wat ze zijn. De tweede functie zou het begin van een ERROR-handler kunnen zijn. Ze wordt opgeroepen bij een error en zal opnieuw terugspringen naar het top-level na het printen van een of andere boodschap.

CATCH and THROW zijn niet zonder gevaar. Soms is het nodig om allerlei zaken af te handelen alvorens de exit uit te voeren. Hier is een voorbeeld. We willen een file-systeem implementeren waar onder meer de volgende activiteit gebeurt:

```
(PROGN
  (OPEN-FILE)
  (DOE-IETS-MET-FILE)
  (SLUIT-FILE))
```

Het zou kunnen dat om de een of andere reden er iets misloopt tijdens het uitvoeren van DOE-IETS-MET-FILE. Code binnen DOE-IETS-MET-FILE zal terugvallen via een throw naar een error-handler. Voor dit gebeurt willen we echter zeker de file terug sluiten.

LISP heeft een extra constructie om aan deze situatie het hoofd te bieden: UNWIND-PROTECT.

```
(UNWIND-PROTECT beschermde-formule noodzakelijke-formules)
```

zorgt ervoor dat als er iets misgaat in het evalueren van de beschermde formule, de noodzakelijke formules toch nog uitgevoerd worden.

Het gegeven voorbeeld wordt dan

```
(PROGN
  (OPEN-FILE)
  (UNWIND-PROTECT
     (DOE-IETS-MET-FILE)
   (SLUIT-FILE)))
```

De file zal nu altijd gesloten worden, zelfs al gebeurt er een non-locale exit vanuit DOE-IETS-MET-FILE.

8. SAMENVATTING

In dit deel hebben we het basisrepertorium uitgebreid met primitieve functies voor getallen en strings en constructies gezien om imperatief te progammeren in LISP.

De belangrijkste rekenkundige functies zijn +, *, /, -. =, >, <, EVENP en ODDP zijn predikaten om getallen te vergelijken.

LOG, SIN, SIND, COS, COSD, ATAN, ATAN2 zijn trigonometrische functies.

SYMBOL-NAME levert de pname van een symbool, INTERN maakt een symbool met een gegeven string als pname, en STRING-LESSP is en predikaat dat nagaat of twee strings in alfabetische volgorde staan. GENSYM genereert een nieuw symbool.

PROG is een functie die toelaat een programma in imperatieve stijl te construeren. PROG heeft hulpmiddelen voor het declareren van variabelen (de prog-variabelen),

het aanduiden en terugkeren naar een bepaald punt (via een label en GO), en het beëindigen van de iteratie met een bepaalde waarde (RETURN). DO is een syntactisch equivalent voor PROG.

Globale variabelen zijn bekend in alle functies en kunnen ook door alle functies veranderd worden met SETQ en SET. DEFVAR en DEFCONSTANT dienen om globale variabelen te declareren.

Destructieve operaties over lijsten veranderen intern de representatie. RPLACA verandert het eerste element. RPLACD verandert de cdr. NCONC voegt twee lijsten bij elkaar.

CATCH, THROW en UNWIND-PROTECT dienen om ongestructureerde controlestructuren op te bouwen.

9. GEMENGDE OPGAVEN

Zoals in het vorige deel bekijken we nu een aantal problemen van grotere omvang. Eerst de opgaven, dan de oplossingen en tenslotte nog enkele opgaven die niet worden opgelost.

9.1. DE OPGAVEN

9.1.1. VAN BINAIR NAAR DECIMAAL

Schrijf functies om een positief geheel getal om te zetten naar een binaire representatie en omgekeerd. Een binair getal is voorgesteld als een lijst van 0 en 1.

Een bekend algoritme om het binair equivalent van een decimaal geheel getal te vinden gaat als volgt:

1. Bereken de rest van het getal en 2 en voeg dit toe aan het binaire getal.
2. Bereken het vervolg van het binaire getal door stap 1 te herhalen voor het getal gedeeld door 2.
3. Stop als het getal gelijk is aan 0.

Bijvoorbeeld voor 5 krijgen we

```
(REMAINDER 5 2) = 1; (/ 5 2) = 2
(REMAINDER 2 2) = 0; (/ 2 2) = 1
(REMAINDER 1 2) = 1; (/ 1 2) = 0
```

Het binaire equivalent van 5 is daarom gelijk aan 101.

Een even bekend algoritme om het decimale equivalent van een binair getal te berekenen gaat als volgt. Veronderstel N gelijk aan 0.

1. Vermenigvuldig N met 2 en tel het eerste getal uit de binaire representatie erbij op.
2. Herhaal deze operatie iteratief voor elk getal.

N is het gezochte getal. Bijvoorbeeld, 101 is gelijk aan 5 omdat

```
N = 0;
(* 0 2) + 1 = 1;
(* 1 2) + 0 = 2;
(* 2 2) + 1 = 5.
```

Schrijf eerst een applicatieve, recursieve functie. Herschrijf dan de functies met DO.

9.1.2. BELSORTERING

Een lijst getallen moet gerangschikt worden volgens grootte, het kleinste getal eerst. Een niet erg efficiënte methode om dit te doen is door telkens twee getallen te vergelijken en ze te verwisselen van plaats als het eerste groter is dan het tweede. Bijvoorbeeld voor de lijst (5 3 1 2) krijgen we

```
(5 3 1 2)
(3 5 1 2)
(3 1 5 2)
(3 1 2 5)
```

Merk op dat het grootste getal zich stapsgewijze naar het einde verplaatst. De methode kan dan herhaald worden om het op één na grootste getal op zijn plaats te krijgen en zo verder tot alle getallen op hun plaats staan. Dit lijkt op luchtbellen die in het water opstijgen, vandaar de naam belsortering (in het Engels 'bubblesort'). Schrijf een functie die een lijst getallen rangschikt volgens deze methode. Gebruik destructieve operaties voor het veranderen van plaats en een DO-lus voor de iteratie.

9.1.3. EEN WOORDENBOEK

Ontwerp een woordenboek in LISP. Gebruik een binaire boomstructuur. Elke knoop in de boom bevat een woord. Links komen de woorden die lager komen in het alfabet, rechts de woorden die hoger komen. Een boom kan worden voorgesteld als een lijst met drie elementen: de knoop, een lijst voor de

linkerdeelboom en een lijst voor de rechterdeelboom. Als een deelboom leeg is, is die gelijk aan NIL. Bijvoorbeeld:

```
(ADAM (ABRAHAM) (EVA))
```

is een mini-woordenboek. De topknoop bevat ADAM. ABRAHAM is de top van de linkerdeelboom. EVA is de topknoop van de rechterdeelboom.

Schrijf functies voor het toevoegen en opzoeken van een woord. De functie die woorden toevoegt, verandert destructief het woordenboek. Het woordenboek zelf is opgeslagen op een globale variabele (bijvoorbeeld WB) die in het begin gelijk gezet wordt aan NIL.

Hier zijn enkele interacties en de stand van het woordenboek na elke interactie:

```
(SETQ WB NIL)
WB = NIL

(VOEGTOE 'SLIM)
WB = (slim)

(VOEGTOE 'DOM)
WB = (slim (dom) nil)

(VOEGTOE 'OLIEDOM)
WB = (slim (dom nil (oliedom)) nil)

(VOEGTOE 'SLIMMER)
WB = (slim (dom nil (oliedom)) (slimmer))

(VOEGTOE 'OLIEDOM)
T, het woordenboek blijft gelijk

(VOEGTOE 'TREIN)
WB = (slim (dom nil (oliedom)) (slimmer nil (trein)))
```

9.2. DE OPLOSSINGEN

9.2.1. VAN BINAIR NAAR DECIMAAL EN TERUG

Eerst een recursieve functie om een positief geheel getal om te zetten naar een binaire representatie:

```
(DEFUN GEHEEL-BINAIR (GETAL)
  (COND
    ((= GETAL 0) (LIST 0))
    (T (APPEND
         (GEHEEL-BINAIR (/ GETAL 2))
         (LIST (REMAINDER GETAL 2))))))
```

Een equivalente definitie met DO.

```
(DEFUN GEHEEL-BINAIR2 (GETAL)
  (DO*
    ((M GETAL (/ M 2))
     (N NIL (CONS (REMAINDER M 2) N)))
    ((= M 0) N)))
```

Nu een recursieve functie om een binair getal om te zetten in decimaal. We gebruiken een hulpfunctie AUX-BINAIR-GEHEEL die via staartrecursie het getal berekent.

```
(DEFUN BINAIR-GEHEEL (GETAL)
  (AUX-BINAIR-GEHEEL GETAL 0))

(DEFUN AUX-BINAIR-GEHEEL (GETAL N)
  (COND
    ((NULL GETAL) N)
    (T (AUX-BINAIR-GEHEEL
         (CDR GETAL)
         (+ (* N 2) (CAR GETAL))))))
```

Een equivalente definitie met DO:

```
(DEFUN BINAIR-GEHEEL2 (GETAL)
  (DO*
    ((M GETAL (CDR M))
     (N 0 (+ (* N 2) (CAR M))))
    ((NULL M) N)))
```

9.2.2. BELSORTERING

Laten we eerst een functie BELSORTERING1 definiëren die het grootste element op zijn plaats brengt. Er zijn vier gevallen:

1. De lijst die gesorteerd moet worden is de lege lijst. Dan is het resultaat uiteraard gelijk aan NIL.

2. De lijst bevat slechts 1 element. Dan is het resultaat gelijk aan de lijst zelf.
3. Het eerste element in de lijst is kleiner dan het tweede element. Dan passen we de functie toe op de rest van de lijst.
4. Het eerste element is groter. Dan moeten we de elementen omdraaien en de functie recursief opnieuw toepassen.

Omgezet in LISP is dit:

```
(DEFUN BELSORTERING1 (LIJST)
  (COND
    ((NULL LIJST) NIL)
    ((NULL (CDR LIJST)) LIJST)
    ((NOT (> (CAR LIJST) (CADR LIJST)))
      (CONS
        (CAR LIJST)
        (BELSORTERING1 (CDR LIJST))))
    (T
      (LET
       ((EERSTE-ELEMENT (CAR LIJST)))
        (RPLACA LIJST (CADR LIJST))
        (RPLACA (CDR LIJST) EERSTE-ELEMENT))
       (BELSORTERING1 (CDR LIJST)))))
```

Nu introduceren we een globale variabele VERANDERING die bijhoudt of er veranderingen zijn opgetreden. BELSORTERING1 verandert deze variabele van NIL naar T als het een verandering uitvoert:

```
(DEFUN BELSORTERING1 (LIJST)
  (COND
    ((NULL LIJST) NIL)
    ((NULL (CDR LIJST)) LIJST)
    ((NOT (> (CAR LIJST) (CADR LIJST)))
      (CONS (CAR LIJST)
            (BELSORTERING1 (CDR LIJST))))
    (T
     (SETQ VERANDERING T)
     (LET
       ((EERSTE-ELEMENT (CAR LIJST)))
       (RPLACA LIJST (CADR LIJST))
       (RPLACA (CDR LIJST) EERSTE-ELEMENT))
      (BELSORTERING1 (CDR LIJST)))))
```

BELSORTERING zelf roept BELSORTERING1 op tot er geen veranderingen meer mogelijk zijn:

```
(DEFUN BELSORTERING (LIJST)
  (DO ()
      ((NULL VERANDERING) LIJST)
    (SETQ VERANDERING NIL)
    (BELSORTERING1 LIJST)))
```

9.2.3. EEN WOORDENBOEK

Het woordenboek wordt gedeclareerd als een globale variabele WB:

```
(DEFVAR WB)
```

Eerst de functie VOEGTOE die een woord toevoegt aan het woordenboek. Als het woordenboek gelijk is aan NIL, dan zet VOEGTOE het woordenboek gelijk aan een lijst met als element dit woord. Dit woord is immers de topknoop van het woordenboek. Anders past VOEGTOE een hulpfunctie VOEGTOE2 toe:

```
(DEFUN VOEGTOE (WOORD)
  (COND
    ((NULL WB) (SETQ WB (LIST WOORD)))
    (T (VOEGTOE2 WOORD WB))))
```

VOEGTOE2 zoekt recursief naar het woord in het woordenboek. Als het woord gelijk is aan de topknoop van de boom, is het al aanwezig. Als het woord alfabetisch kleiner is dan de topknoop zoekt VOEGTOE2 verder in de linkerknoop via de functie VOEGTOE-LINKERKNOOP. Als het woord alfabetisch groter is, zoekt het via de functie VOEGTOE-RECHTERKNOOP:

```
(DEFUN VOEGTOE2 (WOORD WB)
  (COND
    ((EQ (CAR WB) WOORD) T)
    ((ALPHALESSP WOORD (CAR WB))
     ;; voeg toe aan linkerknoop
     (VOEGTOE-LINKERKNOOP WOORD WB))
    (T
     ;; voeg toe aan rechterknoop
     (VOEGTOE-RECHTERKNOOP WOORD WB))))
```

VOEGTOE-LINKERKNOOP kijkt of er elementen zijn na de topknoop. Als er geen deelbomen zijn, kan het nieuwe woord direct worden toegevoegd. Als er nog geen elementen zijn in de linkerdeelboom, moet een nieuwe deelboom worden

begonnen. Anders roept VOEGTOE-LINKERKNOOP recursief VOEGTOE2 op die nagaat of het woord in de linkerknoop zit.

```
(DEFUN VOEGTOE-LINKERKNOOP (WOORD WB)
    (COND
      ((NULL (CDR WB))
       ;; er zijn geen deelbomen
       (RPLACD WB (LIST (LIST WOORD) NIL)))
      ((NULL (CADR WB))
       ;; de linkerdeelboom is leeg
       (RPLACA (CDR WB) (LIST WOORD)))
      (T
       (VOEGTOE2 WOORD (CADR WB)))))
```

VOEGTOE-RECHTERKNOOP is analoog aan VOEGTOE-LINKER-KNOOP:

```
(DEFUN VOEGTOE-RECHTERKNOOP (WOORD WB)
    (COND
      ((NULL (CDR WB))
       ;; er zijn geen deelbomen
       (RPLACD WB (LIST NIL (LIST WOORD))))
      ((NULL (CADDR WB))
       ;; de rechterdeelboom is leeg
       (RPLACA (CDDR WB) (LIST WOORD)))
      (T
       (VOEGTOE2 WOORD (CADDR WB)))))
```

Nu de functie ZOEKOP. Deze functie is een eenvoudige recursieve functie:

```
(DEFUN ZOEKOP (WOORD WB)
  (COND
    ((NULL WB)
     ;; het woordenboek is leeg
     NIL)
    ((EQ (CAR WB) WOORD)
     ;; de topknoop is gelijk aan het woord
     T)
    ((NULL (CDR WB))
     ;; er zijn geen deelbomen
     NIL)
    ((ALPHALESSP WOORD (CAR WB))
     ;; zoek in linkerknoop
     (ZOEKOP WOORD (CADR WB)))
    (T
     ;; zoek in rechterknoop
     (ZOEKOP WOORD (CADDR WB)))))
```

9.3. NOG MEER OPGAVEN

1. Belsortering kan efficiënter gemaakt worden door bij te houden waar de laatste verandering heeft plaats gegrepen zodanig dat vanaf dat punt geen vergelijkingen meer moeten gebeuren. Wijzig de functies voor belsortering in deze zin.
2. Herschrijf BELSORTERING zodanig dat de globale variabele VERANDERING niet langer globaal is.
3. Schrijf een functie die woorden wegdoet uit het woordenboek.
4. Schrijf functies die getallen van een decimale representatie omzetten naar een octale representatie.

DEEL 4. FUNCTIONEEL PROGRAMMEREN

Dit deel bestudeert LISP-functies. We beginnen met mechanismen om de argumenten van een functie nauwkeuriger te definiëren. Dan volgen macro's die toelaten om een uitdrukking vóór evaluatie om te zetten in een andere uitdrukking. Tenslotte bestuderen we functies die functies als argumenten hebben en functies van hogere orde. Deze maken het mogelijk om in LISP functies als volwaardige objecten te gebruiken en op die manier programma's te schrijven.

1. DEFINITIE VAN ARGUMENTEN

1.1. DEFINITIES

Er zijn diverse faciliteiten om de argumenten van een functie beter te omschrijven. Dit gebeurt via speciale symbolen die voor de naam van de variabele komen. Deze symbolen beginnen steeds met het teken &.

&OPTIONAL duidt aan dat het argument dat volgt optioneel is. Eventueel kan er een default-waarde via een formule worden aangegeven. In dit geval wordt de naam van de variabele en de formule in een lijst gegroepeerd. Bijvoorbeeld, gegeven de definitie

```
(DEFUN FOO (Y &OPTIONAL (X 1))
  (+ Y X))
```

dan kan FOO opgeroepen worden met één argument:

```
(F 4)
```

wat 5 oplevert omdat X gebonden is aan 1. Of het kan opgeroepen worden met twee argumenten

```
(FOO 4 5)
```

wat 9 oplevert.

Een functie kan meer dan één optioneel argument hebben maar ze worden natuurlijk van links naar rechts gebonden.

&REST duidt aan dat alle resterende objecten bij de oproep van de functie moeten worden gezien als één lijst. De variabele die volgt op &REST wordt gebonden aan deze lijst. Hier is bijvoorbeeld een definitie van de functie LIST:

```
(DEFUN LIST (&REST ARGS)
  ARGS)
```

Beschouw nu de oproep

```
(LIST 1 5 '(A B))
```

dan zal ARGS gebonden worden aan het resultaat van de doorgegeven elementen gegroepeerd in een lijst, d.w.z. (1 5 (A B)).

Het is tenslotte mogelijk om met diverse variabelen sleutelwoorden te associëren wat het duidelijker maakt waarvoor een variabele dient. Het maakt het ook mogelijk om de volgorde van de variabelen te veranderen bij de oproep.

De declaratie van sleutelwoorden gebeurt door het symbool &KEY te gebruiken bij de definitie van de argumenten van de functie. De naam van de variabele wordt meteen beschouwd als het sleutelwoord voor deze variabele. Als men een ander sleutelwoord wil dan de naam van de variabele, kan dit door een paar in te geven met eerst het sleutelwoord en dan de naam van de variabele.

Bijvoorbeeld:

```
(DEFUN IS-LID (&KEY ELEMENT LIJST)
  (MEMBER ELEMENT LIJST))
```

zorgt ervoor dat ELEMENT en LIJST als sleutelwoorden kunnen worden gebruikt.

```
(DEFUN IS-LIST (&KEY ((:ELEMENT X)) ((:LIJST Y)))
  (MEMBER X Y))
```

geeft dezelfde sleutelwoorden. De variabelen X en Y zijn nu de namen die intern in de functie worden gebruikt. Sleutelwoorden worden gewoonlijk vooraf gegaan door een ':'. Meer hierover in hoofdstuk 5.6 dat packages behandelt.

Een oproep van de functie IS-LID kan nu gebeuren met

```
(IS-LID :ELEMENT '(A B C) :LIJST '((A B C)))
```

:ELEMENT en :LIJST zijn sleutelwoorden. Omdat sleutelwoorden expliciet aangeven welke rol elk argument speelt kunnen we ook zeggen

```
(IS-LID :LIJST '((A B C)) :ELEMENT '(A B C))
```

Vooral zeer grote LISP systemen die door een team van programmeurs worden gebouwd maken uitvoerig gebruik van sleutelwoorden. Ook de meeste systeemfuncties hebben sleutelwoorden voor hun argumenten.

1.2. OEFENINGEN

1. Gegeven

```
(DEFUN FOO (&OPTIONAL (EERSTE 1) (TWEEDE 2))
  (+ EERSTE TWEEDE))
```

Wat is het resultaat van (+ (FOO) (FOO 2) (FOO 3 4))?

Het resultaat is 14. (FOO) geeft (+ 1 2) omdat EERSTE dan aan 1 en TWEEDE dan aan 2 gebonden is. (FOO 2) geeft (+ 2 2) omdat EERSTE nu gelijk is aan de gegeven waarde 2, en TWEEDE de optionele waarde 2 krijgt. Tenslotte (FOO 3 4) is gelijk aan 7 omdat nu beide argumenten gebonden zijn.

2. Veronderstel even dat + slechts twee argumenten kan hebben. Schrijf een functie MANY-PLUS die een onbepaald aantal argumenten bij elkaar optelt. Bijvoorbeeld (MANY-PLUS 1 2 3 4) ⇒ 10.

```
(DEFUN MANY-PLUS (&REST ARGS)
  (RECURSIVE-PLUS ARGS))

(DEFUN RECURSIVE-PLUS (ARGS)
  (COND
    ((NULL ARGS) 0)
    (T (+ (CAR ARGS) (RECURSIVE-PLUS (CDR ARGS))))))
```

3. Definieer een functie KLEINER-DAN die kan opgeroepen worden met de sleutelwoorden :KLEINSTE en :GROOTSTE, zoals in (KLEINER-DAN :KLEINSTE X :GROOTSTE Y)

Een eerste mogelijkheid is

```
(DEFUN KLEINER-DAN (&KEY KLEINSTE GROOTSTE)
  (< KLEINSTE GROOTSTE))
```

Een tweede mogelijkheid is

```
(DEFUN KLEINER-DAN (&KEY ((:KLEINSTE A))
                         ((:GROOTSTE B)))
  (< A B))
```

2. MACRO'S

2.1. DEFINITIES

Het is soms mogelijk om substituties uit te voeren op formules zonder de argumenten te evalueren. Dit is interessant omdat de compiler deze substituties kan uitvoeren vóór de definitie om te zetten. Op deze manier verdwijnt de overhead van de extra functie-oproep.

Veronderstel bijvoorbeeld de functie TWEEDE:

```
(DEFUN TWEEDE (X)
  (CAR (CDR X)))
```

Als de evaluator de uitdrukking (TWEEDE '(A B C)) toegespeeld krijgt, zal hij de definitie (CAR (CDR X)) evalueren met X gelijk aan '(A B C). De extra evaluatiestap kan in principe vermeden worden door de formule (TWEEDE '(A B C)) direct om te zetten in (CAR (CDR '(A B C))). Deze substitutie kan gebeuren in functies die TWEEDE gebruiken, *voor* ze gecompileerd worden. In de gecompileerde versie is de extra evaluatie-stap dan volledig verdwenen. Er is dus geen extrakost voor het gebruik van de meer leesbare functie TWEEDE.

Een MACRO is een functie die gedefinieerd is in termen van een uitdrukking die moet gesubstitueerd worden voor de formule waarin de macro voorkomt. In plaats van DEFUN gebruiken we DEFMACRO.

Bijvoorbeeld, TWEEDE gedefinieerd als macro ziet er zo uit:

```
(DEFMACRO TWEEDE (X)
  (LIST 'CAR (LIST 'CDR X)))
```

De definitie zegt hoe de formule die in de plaats moet komen van de originele formule eruit ziet. Bijvoorbeeld voor X gelijk aan '(A B C), is (LIST 'CAR (LIST 'CDR X)) gelijk aan (CAR (CDR '(A B C))). Dit moet dan in de plaats komen van (TWEEDE '(A B C)).

Met backquote kan een functie gemakkelijk in een macro getransformeerd worden. Definieer eerst de functie alsof ze geen macro is. Voeg dan een backquote toe en de nodige komma's. Bijvoorbeeld de definitie van TWEEDE is

```
(DEFUN TWEEDE (X)
  (CAR (CDR X)))
```

als macro wordt dit

```
(DEFMACRO TWEEDE (X)
  '(CAR (CDR ,X)))
```

Uiteraard is het op deze manier niet mogelijk om een recursieve functie te definiëren als een macro omdat dan de substitutie oneindig zou blijven duren! Dus alleen substitutiefuncties kunnen omgezet worden in macro's. Het is echter wel mogelijk om andere macrofuncties te gebruiken in een macro zelf.

Merk op dat de argumenten van een macro ook weer gespecialiseerd kunnen worden met &OPTIONAL, &REST of &KEY.

2.2. OEFENINGEN

1. Definieer een macro voor DERDE.

```
(DEFMACRO DERDE (X)
  '(CAR (CDR (CDR ,X)))).
```

2. Waarin wordt (DERDE '(A B C)) omgezet?

```
(CAR (CDR (CDR '(A B C))))
```

3. Definieer een macro voor de functie IF.

```
(DEFMACRO IF (P E1 E2)
  '(COND
     (,P ,E1)
     (T ,E2)))
```

4. Wat is de substitutie van (IF (NUMBERP 5) 10 15)?

```
(COND ((NUMBERP 5) 10) (T 15))
```

5. Schrijf een functie QLIST die een lijst maakt van zijn argumenten. De argumenten worden niet geëvalueerd. Bijvoorbeeld (QLIST (A B C) D E) ⇒ ((A B C) D E).

Dit is een macro:

```
(DEFMACRO QLIST (&REST ARGS)
  ARGS)
```

In het algemeen moeten alle functies die hun argumenten niet evalueren als macro's gedefinieerd worden.

6. Construeer een 'slimme plus' die al een deel van de bewerking doet vóór evaluatie. Bijvoorbeeld (SLIMME-PLUS X 0) reduceert onmiddellijk naar X.

```
(DEFMACRO SLIMME-PLUS (ARG1 ARG2)
  (COND
    ((= ARG1 0) ARG2)
    ((= ARG2 0) ARG1)
    (T '(+ ,ARG1 ,ARG2))))
```

7. Schrijf een functie die een PASCAL-stijl WHILE implementeert via PROG. De functie is van de vorm (WHILE P $F_1 \cdots F_n$). De formules $F_1 \cdots F_n$ worden uitgevoerd tot het predikaat P onwaar oplevert. Bijvoorbeeld,

```
(PROG (X)
  (SETQ X 5)
  (WHILE (> X 0)
    (PRINT X)
    (SETQ X (1- X)))
  (RETURN X))
```

print de getallen 5 4 3 2 1.

```
(DEFMACRO WHILE (PREDIKAAT &REST FORMULES)
  '(EXECUTE-WHILE ,PREDIKAAT ,FORMULES))
```

```
(DEFUN EXECUTE-WHILE (PREDIKAAT FORMULES)
  (PROG ()
   BEGIN
    ;; evaluatie van de formules begint
    (PROG (F)
      (SETQ F FORMULES)
     BEGIN2
      (EVAL (CAR F))
      (SETQ F (CDR F))
      (COND
        ((NULL F) (RETURN T))
        (T (GO BEGIN2)))))
    ;; einde van de evaluatie van de formules
    (COND
      ((NOT (EVAL PREDIKAAT)) (RETURN NIL))
      (T (GO BEGIN)))))
```

8. Definieer een functie die een PASCAL-stijl REPEAT-UNTIL constructie implementeert zodat het volgende programma mogelijk wordt:

```
(PROG (X)
  (SETQ X 5)
  (REPEAT
    (PRINT X)
    (SETQ X (1- X))
   UNTIL
    (= X 0))
  (RETURN X))
```

```
(DEFMACRO REPEAT (&REST FORMULES)
  (LET
    ((PREDIKAAT (LAATSTE FORMULES)))
     (REPEAT-FORMULES
       (ALL-BUT-2 FORMULES)))
    '(DO ()
         (,PREDIKAAT)
       ,@REPEAT-FORMULES)))
```

LAATSTE geeft het laatste element in een lijst terug, ALL-BUT-2 geeft dezelfde lijst terug, maar zonder de twee laatste elementen.

```
(DEFUN LAATSTE (LIJST)
  (IF (NULL (CDR LIJST))
      (CAR LIJST)
      (LAATSTE (CDR LIJST))))

(DEFUN ALL-BUT-2 (LIJST)
  (REVERSE
    (CDDR
      (REVERSE LIJST))))
```

3. FUNCTIES ALS ARGUMENTEN

3.1. DEFINITIES

We hebben in oefening 6 van hoofdstuk 3.1. GEMIDDELDE als volgt gedefinieerd:

```
(DEFUN GEMIDDELDE (GETALLENREEKS)
  (/
    (SOM GETALLENREEKS)
    (LENGTH GETALLENREEKS)))
```

waarbij

```
(DEFUN SOM (GETALLENREEKS)
  (COND
    ((NULL GETALLENREEKS) 0)
    (T
      (+
        (CAR GETALLENREEKS)
        (SOM (CDR GETALLENREEKS))))))
```

We zouden echter een kortere definitie van GEMIDDELDE kunnen schrijven als we + konden gebruiken met GETALLENREEKS als lijst van argumenten. Immers + neemt een onbepaald aantal argumenten en telt ze op. Dus als de getallenreeks gelijk is aan (1 2 3) dan is (+ 1 2 3) gelijk aan 6. Merk op dat we niet kunnen zeggen (+ GETALLENREEKS) want getallenreeks is een lijst en geen getal. + verwacht getallen als argumenten, geen lijsten.

+ kan wel direct toegepast worden aan de hand van de functie APPLY. APPLY heeft twee argumenten: een functie en een lijst van de argumenten voor deze functie:

```
(APPLY functie argumenten)
```

APPLY past de functie toe op de argumenten. Bijvoorbeeld

```
(APPLY '+ '(1 2 3))
```

is gelijk aan 6 en equivalent aan

```
(+ 1 2 3)
```

Hier is een nieuwe definitie van GEMIDDELDE die gebruik maakt van APPLY:

```
(DEFUN GEMIDDELDE (GETALLENREEKS)
  (/
    (APPLY '+ GETALLENREEKS)
    (LENGTH GETALLENREEKS)))
```

Merk op dat APPLY (zoals verwacht) enkel zijn twee argumenten evalueert, en er dus geen verdere evaluatie van de elementen uit de lijst gebeurt voor ze aan de functie worden doorgegeven. Dus de evaluatie van

```
(APPLY 'CONS '((+ 2 3) (4)))
```

is gelijk aan ((+ 2 3) 4) en niet (5 4)!

FUNCALL is een functie die hetzelfde doet als APPLY, dit wil zeggen een functie toepassen op een reeks argumenten, behalve dat FUNCALL een onbepaald aantal argumenten heeft waarvan het eerste argument de functie en de overige argumenten de argumenten zijn vóor deze functie:

(FUNCALL functie arg_1 · · · arg_n)

Merk hier ook op dat (opnieuw zoals verwacht) FUNCALL eerst *alle* argumenten evalueert alvorens de functie toe te passen. Dus

```
(FUNCALL 'CONS (+ 2 3) '(4))
```

is gelijk aan (5 4).

APPLY en FUNCALL zijn voorbeelden van functies die een andere functie als argument hebben. Een ander voorbeeld vormen de MAP-functies die een functie herhaaldelijk toepassen op een reeks van argumenten.

Een eerste voorbeeld van een map-functie is MAPCAR. MAPCAR heeft een onbepaald aantal argumenten. Het eerste argument is een functie. De overige argumenten zijn lijsten van argumenten voor deze functie:

(MAPCAR functie arg_1 · · · arg_n)

Deze lijsten functioneren als stapels. MAPCAR neemt telkens een argument van elke stapel en past de functie hierop toe. Bovendien wordt het resultaat van elke oproep gegroepeerd in een lijst. Als de lijst van argumenten NIL is, levert MAPCAR NIL op. Zodra één van de lijsten uitgeput is, stopt MAPCAR en houdt geen rekening meer met eventuele resten van andere lijsten. Dus de evaluatie van

(MAPCAR '+ '(1 2) '(3 4 5))

leidt tot de evaluatie van (+ 1 3) en (+ 2 4) en tot de groepering in een lijst van deze deelresultaten. Het eindresultaat is (4 6).

Als de functie opgeroepen in MAPCAR slechts één argument heeft is er uiteraard slechts één stapel van argumenten nodig. Bijvoorbeeld:

(MAPCAR '1+ '(1 2 3))

is gelijk aan (2 3 4). Hier is nog een voorbeeld

(MAPCAR 'CAR '(((A)) ((B))))

leidt tot de evaluatie van (CAR '((A))) en (CAR '((B))) en tot het eindresultaat ((A) (B)).

LISP bevat een hele reeks mapping functies afhankelijk van de manier waarop de lijst doorlopen wordt of afhankelijk van de manier waarop het resultaat weer wordt samengevoegd. Hier zijn enkele van de belangrijkste:

1. (MAPCAR functie arg_1 · · · arg_n)
   selecteert de elementen uit de lijsten gegeven als arg_1, ... , arg_n met CAR, en voegt het resultaat van het toepassen van de functie op elk van deze elementenreeksen bij elkaar via CONS.

2. (MAPLIST functie arg_1 · · · arg_n)
   selecteert de elementen uit de lijsten gegeven als arg_1, ... , arg_n met CDR, en voegt het resultaat van het toepassen van de functie op elk van deze elementenreeksen bij elkaar via CONS. Veronderstel bijvoorbeeld (MAPLIST 'CAR '(1 2 3 4)). Het resultaat is (1 2 3 4) want:

   (CAR '(1 2 3 4)) ⇒ 1
   (CAR '(2 3 4)) ⇒ 2
   (CAR '(3 4)) ⇒ 3
   (CAR '(4)) ⇒ 4

3. (MAPCAN functie arg_1 · · · arg_n)
selecteert de elementen uit de lijsten gegeven als arg_1, ... , arg_n met CAR, en voegt het resultaat van het toepassen van de functie op elk van deze elementenreeksen bij elkaar via APPEND[1]. Bijvoorbeeld, (MAPCAN 'CAR '(((A)) ((B)))) is gelijk aan (A B), immers (CAR '((A))) is gelijk aan (A) en (CAR '((B))) is gelijk aan (B) en (APPEND '(A) '(B)) is gelijk aan (A B).

Merk op dat mapping functies een alternatieve manier zijn om een iteratie uit te voeren.

3.2. OEFENINGEN

1. Veronderstel een functie 1-. Schrijf een functie SUB1-IN-LIJST die 1 aftrekt van elk element in een lijst. Gebruik MAPCAR.

```
(DEFUN SUB1-IN-LIJST (L)
  (MAPCAR '1- L))
```

2. Schrijf dezelfde functie recursief zonder MAPCAR.

```
(DEFUN SUB1-IN-LIJST (L)
  (COND
    ((NULL L) NIL)
    (T
      (CONS
        (1- (CAR L))
        (SUB1-IN-LIJST (CDR L))))))
```

3. Schrijf een functie MAAK-DEELLIJSTEN die een lijst maakt bestaande uit alle deellijsten van een lijst. Bijvoorbeeld (MAAK-DEELLIJSTEN '(A B C)) is gelijk aan '((A B C) (B C) (C)). Hint: Maak eerst een identiteitsfunctie en gebruik MAPLIST.

We definiëren de identiteitsfunctie, d.w.z. een functie die zijn argument gewoon teruggeeft.

[1] In feite is het met NCONC, een destructieve versie van APPEND.

```
(DEFUN IDENTITY (X) X)
```

Het probleem is nu een triviale toepassing van MAPLIST:

```
(DEFUN MAAK-DEELLIJSTEN (LIJST)
  (MAPLIST 'IDENTITY LIJST))
```

MAPLIST gaat telkens de CDR nemen van de lijst en het resultaat bijeen brengen in een nieuwe lijst.

4. Schrijf een functie VERWIJDER-HAAKJES die gegeven een lijst van sublijsten, de elementen van elke sublijst samenbrengt in een nieuwe lijst. Bijvoorbeeld (VERWIJDER-HAAKJES '((A B C) (D E F))) levert (A B C D E F) op.

Dit kan eenvoudig gebeuren met de functie MAPCAN en de identiteitsfunctie:

```
(DEFUN VERWIJDER-HAAKJES (LIJST)
  (MAPCAN 'IDENTITY LIJST))
```

5. Schrijf een functie die nagaat of een lijst het element zero bevat. Veronderstel het predicaat ZEROP dat T oplevert als het argument zero is. Gebruik een mapping functie.

Het eerste wat we doen is de lijst omzetten in een lijst van waarheidswaarden. Via APPLY kan dan de functie OR toegepast worden op deze lijst.

```
(DEFUN ZEROP-IN-LIJST (LIJST)
  (APPLY 'OR (MAPCAR 'ZEROP LIJST)))
```

Veronderstel bijvoorbeeld (ZEROP-IN-LIJST '(A B C 0 1)). Evaluatie van (MAPCAR 'ZEROP '(A B C 0 1)) levert (NIL NIL NIL T NIL). (APPLY 'OR '(NIL NIL NIL T NIL)) levert T op.

6. Nog een voorbeeld van een primitieve LISP-functie die een andere functie als argument heeft is de functie SORT. SORT heeft twee argumenten, een lijst waarvan de elementen moeten gerangschikt worden, en een predikaat dat beschrijft hoe de rangschikking moet gebeuren:

```
(SORT lijst functie)
```

Het predikaat moet T of NIL opleveren bij het vergelijken van twee elementen. Bijvoorbeeld,

```
(SORT '(5 1 4 3) '>)
```

is gelijk aan (5 4 3 1). Schrijf een functie ALFABETISEER die een lijst van symbolen alfabetisch rangschikt. Veronderstel het predicaat STRING-LESSP dat nagaat of twee symbolen alfabetisch voor elkaar komen.

```
(DEFUN ALFABETISEER (SYMBOLEN)
  (SORT SYMBOLEN 'STRING-LESSP))
```

7. Aan welke voorwaarde moet het predikaat in SORT voldoen?

Als de argumenten van het predikaat veranderen van plaats moet het predikaat veranderen van waarde.

8. Definieer de functie FUNCALL met APPLY.

FUNCALL heeft een onbepaald aantal argumenten die allemaal worden geëvalueerd:

```
(DEFUN FUNCALL (&REST ARGS)
  (APPLY (CAR ARGS) (CDR ARGS)))
```

9. Niets verhindert ons om ook zelf functies te definiëren die functies als argumenten hebben. Schrijf in deze zin een functie (SUMMATIE F J N) die overeenkomt met de wiskundige notatie:

$$\sum_{i=j}^{n} f(i)$$

Bijvoorbeeld (SUMMATIE 'SQUARE 1 10) berekent de som van de machten van de getallen van 1 tot 10.

```
(DEFUN SUMMATIE (F J N)
  (COND
    ((> J N) 0)
    (T
      (+
        (FUNCALL F J)
        (SUMMATIE F (+ 1 J) N)))))
```

10. De variatie van een reeks getallen $(x_i)_{i=1..n}$ is gelijk aan

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n-1}$$

waarbij $\bar{x}$ het gemiddelde is van de reeks. Definieer deze functie in LISP.

```
(DEFUN STANDAARD-DEVIATIE (GETALLENREEKS)
  (LET
   ((X-BAR (GEMIDDELDE GETALLENREEKS)))
   (/
    (APPLY '+
           (MAPCAR
             '(LAMBDA (X)
               (EXPT (DIFFERENCE X X-BAR) 2))
             GETALLENREEKS))
     (1- (LENGTH GETALLENREEKS)))))
```

11. De mediaan van een getallenreeks is het element dat de $(\frac{n+1}{2})^e$ positie inneemt als we de elementen rangschikken volgens grootte. Definieer de functie MEDIAAN in LISP

NTH (zie deel 1) levert het $(n-1)^e$ element in een lijst en SORT kan gebruikt worden om de elementen te rangschikken volgens grootte.

```
(DEFUN MEDIAAN (GETALLEN)
  (NTH
   (/ (+ (LENGTH GETALLEN) 1) 2)
   (SORT GETALLEN '>)))
```

4. FUNCTIONELE FUNCTIES

4.1. DEFINITIES

Tot hiertoe hebben we steeds de naam van een functie (bijvoorbeeld + of CONS) gebruikt als functioneel argument. Veronderstel nu dat we een functie willen die alleen moet dienen binnen in een MAP-operatie. In dit geval zouden we alleen hiervoor een nieuwe naam moeten introduceren. Beter zou zijn om functies als dusdanig te kunnen definiëren. Dit is mogelijk met de speciale functie LAMBDA,

die dient om dynamisch functies samen te stellen.

De functie LAMBDA heeft twee argumenten: een lijst van variabelen en een formule:

(LAMBDA lijst-van-variabelen $formule_1$ · · · $formule_n$)

Het geheel is een nieuwe functie. Als deze functie wordt toegepast op een reeks argumenten, zorgt LAMBDA ervoor dat de elementen in de lijst-van-variabelen gebonden worden aan de corresponderende argumenten en dat de formules met deze bindingen worden uitgevoerd. Deze operatie heet *lambda-conversie.*

Bijvoorbeeld de volgende uitdrukking is een functie met hetzelfde effect als TWEEDE:

(LAMBDA (X) (CADR X)).

We kunnen deze functie toepassen op (A B C) als volgt:

((LAMBDA (X) (CADR X)) '(A B C)).

LAMBDA bindt X aan (A B C), zodat (CADR X) gelijk is aan B. We kunnen ook zeggen

(APPLY '(LAMBDA (X) (CADR X)) '((A B C)))

of

(FUNCALL '(LAMBDA (X) (CADR X)) '(A B C))

Hier is nog een voorbeeld van het gebruik van LAMBDA:

((LAMBDA (X Y) (APPEND X Y)) '(A B C) '(D E F))

drukt uit dat de lambda-uitdrukking (LAMBDA (X Y) (APPEND X Y)) toegepast moet worden op '(A B C) en '(D E F). Het toepassen van de functie (LAMBDA (X Y) (APPEND X Y)) betekent dat de waarde van X en Y wordt gebonden aan (A B C) en (D E F) in (APPEND X Y). Evaluatie is dan gelijk aan (A B C D E F).

De lambda-functie is impliciet in elke definitie en is daarom de basis van LISP. Bijvoorbeeld als via DEFUN de functie TWEEDE wordt gedefinieerd als:

```
(DEFUN TWEEDE (X)
  (CAR (CDR X)))
```

betekent dit eigenlijk dat de lambda-uitdrukking

(LAMBDA (X) (CAR (CDR X)))

gebonden is aan TWEEDE. Het toepassen van TWEEDE is hetzelfde als het

toepassen van deze lambda-uitdrukking. Bijvoorbeeld

```
(TWEEDE '(A B C))
```

is gelijk aan

```
((LAMBDA (X) (CAR (CDR X))) '(A B C))
```

na substitutie krijgen we

```
(CAR (CDR '(A B C)))
```

en dus B.

De LET-constructie die we in het vorige hoofdstuk gezien hebben is in feite een andere syntactische vorm voor een lambda-uitdrukking:

```
(LET
  ((variabele1 binding1)
   ...)
  formule1  · · ·  formulen)
```

is gelijk aan

```
((LAMBDA (variabele1  · · · ) formule1  · · · ) binding1 · · · )
```

Bijvoorbeeld,

```
(LET
  ((LIJST '(A B C))
   (ELEMENT 'B))
  (CONS ELEMENT LIJST))
```

is equivalent aan

```
((LAMBDA (LIJST ELEMENT) (CONS ELEMENT LIJST))
  '(A B C) 'B)
```

CLOSURES

Dikwijls bevat een lambda-uitdrukking variabelen die gebonden zijn in de omgeving waarin de lambda-uitdrukking voorkomt. Veronderstel bijvoorbeeld dat we een functie definiëren die een nieuwe functie voorbrengt:

```
(DEFUN COMPOSITION (F G)
  '(LAMBDA (X) (FUNCALL F (FUNCALL G X))))
```

De variabelen F en G zullen wel gebonden zijn bij het oproepen van de functie COMPOSITION, maar ze moeten ook gebonden zijn binnen in de lambda-

uitdrukking wanneer deze op zijn beurt wordt toegepast, zoals in

```
(FUNCALL (COMPOSITION 'CAR 'CDR) '(A B))
```

wat gelijk is aan

```
(FUNCALL
  '(LAMBDA (X) (FUNCALL F (FUNCALL G X)))
  '(A B))
```

met F gebonden aan CAR en G gebonden aan CDR. Om deze bindingen te bewaren is het nodig om met de functie een locale omgeving te associëren die de variabelen bevat die in de lambda-uitdrukking voorkomen. Men noemt dit een *closure*. Closures worden gemaakt door de functie FUNCTION. #' is een afkorting van FUNCTION.

COMPOSITION wordt dan

```
(DEFUN COMPOSITION (F G)
  (FUNCTION
    (LAMBDA (X) (FUNCALL F (FUNCALL G X)))))
```

of

```
(DEFUN COMPOSITION (F G)
  #'(LAMBDA (X) (FUNCALL F (FUNCALL G X))))
```

Het veiligste voor beginnende gebruikers is steeds dit speciale teken te plaatsen voor een lambda-uitdrukking.

LAMBDA is een voorbeeld van een functionele functie, dit is een functie die een functie als resultaat heeft. APPLY, FUNCALL, de MAP-functies en LAMBDA geven aanleiding tot een alternatieve stijl van programmeren die noch recursief, noch imperatief is. Neem bijvoorbeeld de functie MEMBER die recursief als volgt is gedefinieerd:

```
(DEFUN MEMBER (ELEMENT LIJST)
  (COND
    ((NULL LIJST) NIL)
    ((EQ ELEMENT (CAR LIJST)) T)
    (T
      (MEMBER ELEMENT (CDR LIJST)))))
```

Een alternatieve definitie is

```
(DEFUN MEMBER (ELEMENT LIJST)
  (APPLY
    'OR
    (MAPCAR
      #'(LAMBDA (X) (EQ ELEMENT X))
      LIJST)))
```

Bijvoorbeeld met het ELEMENT gebonden aan A en LIJST gebonden aan (A B C) zal MEMBER eerst EQ toepassen op A en A, B, C met als resultaat de lijst (T NIL NIL). (OR T NIL NIL) is gelijk aan T, dus het resultaat is T. Het volgende hoofdstuk exploreert deze stijl nog verder.

4.2. OEFENINGEN

1. Schets de evaluatie van

```
((LAMBDA (X Y) (MEMBER X Y)) 'A '(B C ))
```

Substitutie van X en Y levert (MEMBER 'A '(B C)) zodat het eindresultaat gelijk is aan NIL.

2. Schets de evaluatie van

```
((LAMBDA (X)
  ((LAMBDA (Y) (CONS Y NIL)) X))
  'A)
```

X gesubstitueerd in ((LAMBDA (Y) (CONS Y NIL)) X) levert ((LAMBDA (Y) (CONS Y NIL)) 'A). Verdere substitutie van Y levert (CONS 'A NIL). Het eindresultaat is dus (A).

3. Definieer een functie TOON-LAATSTE die gegeven een lijst van symbolen, een corresponderende nieuwe lijst geeft waarbij er 0 staat als het symbool niet het laatste voorkomen is van dat symbool en 1 als het wel het laatste voorkomen is. Bijvoorbeeld (TOON-LAATSTE '(A B C A B C)) levert (0 0 0 1 1 1) op. Gebruik een mapping-functie.

Deze functie is eenvoudig te schrijven met MAPLIST. MAPLIST genereert opeenvolgende deellijsten. We kijken telkens of het eerste element nog voorkomt in de rest van de lijst. Indien ja is het resultaat 0, anders 1.

```
(DEFUN TOON-LAATSTE (LIJST)
  (MAPLIST
    #'(LAMBDA (X)
        (IF (MEMBER (CAR X) (CDR X)) 0 1))
    LIJST))
```

4. Definieer de functie SUBSTITUEER, die een element vervangt door een substituut in een LIJST. Gebruik MAPCAR.

```
(DEFUN SUBSTITUEER (ELEMENT SUBSTITUUT LIJST)
  (MAPCAR
    #'(LAMBDA (X)
        (IF (EQ X ELEMENT) SUBSTITUUT X))
    LIJST))
```

5. Schrijf een uitdrukking met LET die equivalent is aan ((LAMBDA (X) (CONS X NIL)) 'A).

```
(LET ((X 'A)) (CONS X NIL)).
```

6. Wat is het lambda-equivalent van

```
(LET ((X (CAR '(A B C))))
  (APPEND X (CDR X)))
```

```
((LAMBDA (X) (APPEND X (CDR X))) (CAR '(A B C)))
```

7. Definieer een functie ADDER. ADDER heeft één argument, een getal. Het levert een nieuwe functie op die, wanneer toegepast op een ander getal, de twee getallen bij elkaar optelt. Bijvoorbeeld. (FUNCALL (ADDER 1) 2) is gelijk aan 3.

```
(DEFUN ADDER (X) (FUNCTION (LAMBDA (Y) (+ X Y))))
```

of

```
(DEFUN ADDER (X) #'(LAMBDA (Y) (+ X Y)))
```

Merk op dat er wel degelijk een closure moet worden gemaakt omdat de

variabele X moet gebonden blijven binnen de lambda aan de waarde die het had bij de oproep van ADDER.

8. Definieer de functie MAPCAN.

```
(DEFUN MAPCAN (FUNCTIE &REST ARGUMENTLIJSTEN)
  (MAPCAN2 FUNCTIE ARGUMENTLIJSTEN))
```

MAPCAN2 past de functie toe op een lijst bestaande uit het eerste element van elke deellijst en voegt het resultaat met APPEND toe aan de recursieve toepassing van MAPCAN2 op de resten van elke deellijst:

```
(DEFUN MAPCAN2 (FUNCTIE &REST ARGS)
  (COND
   ((OR
      ;; stop als de argumenten uitgeput zijn
      (NULL ARGS)
      (APPLY 'OR (MAPCAR 'NULL ARGS)))
     NIL)
   (T
    (APPEND
     (APPLY
       FUNCTIE
       ;; neem het eerste element van elke deellijst
       (MAPCAR #'(LAMBDA (X) (CAR X)) ARGS))
     (MAPCAN2
       FUNCTIE
       ;; neem de rest van elke deellijst
       (MAPCAR
          #'(LAMBDA (X) (CDR X))
          ARGS))))))
```

9. Gegeven een lijst van lijsten, schrijf een functie FILTER-A die de lijsten oplevert die het element A bevatten. Bijvoorbeeld, (FILTER-A '((A B C) (D E) (A A))) is gelijk aan ((A B C) (A A)).

```
(DEFUN FILTER-A (LIJST)
  (MAPCAN
    #'(LAMBDA (X)
        (IF (MEMBER 'A X) (LIST X) NIL))
    LIJST))
```

10. Definieer de functie LENGTH die de lengte van een lijst berekent met MAPCAR en APPLY.

LENGTH zet eerst de elementen van de lijst om in 1 en past dan + toe:

```
(DEFUN LENGTH (LIJST)
  (APPLY '+ (MAPCAR #'(LAMBDA (X) 1) LIJST)))
```

11. Definieer een functie CONSTANTE die gegeven een argument een nieuwe functie maakt die altijd dit argument oplevert voor gelijk welk argument. Bijvoorbeeld (FUNCALL (CONSTANTE 1) 10) is gelijk aan 1.

```
(DEFUN CONSTANTE (X)
  #'(LAMBDA (Y) X))
```

Bijvoorbeeld (CONSTANTE 1) levert (LAMBDA (Y) 1) op zodanig dat (FUNCALL '(LAMBDA (Y) 1) 10) gelijk is aan 1.

12. Herschrijf LENGTH gebruik makend van CONSTANTE.

```
(DEFUN LENGTH (LIJST)
  (APPLY '+ (MAPCAR (CONSTANTE 1) LIJST)))
```

5. COMBINATOREN EN FUNCTIONEEL PROGRAMMEREN

5.1. DEFINITIES

De vorige hoofdstukken bevatten mechanismen die toelaten functies als volwaardige objecten te gebruiken en te manipuleren. In principe is het mogelijk om programma's te schrijven die uitsluitend bestaan uit operaties over functies en het toepassen van functies op argumenten of reeksen van argumenten. Men noemt deze stijl van programmeren 'functioneel' omdat de programmeur denkt in termen van operaties over functies in de plaats van operaties over objecten zoals lijsten en

getallen. Operaties over functies worden ook *combinatoren* genoemd en de combinatorische logica bestudeert de logische grondslagen van combinatoren. Het is bijvoorbeeld bekend dat een beperkt aantal combinatoren voldoende is om alle mogelijke berekenbare functies te definiëren. Hier zijn enkele voorbeelden van typische combinatoren:

1. (COMPOSITION f_1 · · · f_n) is een functie die een compositie van functies construeert.

```
(FUNCALL (COMPOSITION f1 ··· fn)
  arg1 ··· argn)
```

is gelijk aan

```
(FUNCALL (COMPOSITION f1 ··· fn-1)
  (FUNCALL fn arg1 ··· argn))
```

voor $n > 1$ en

```
(FUNCALL (COMPOSITION f1) arg1 ··· )
```

is gelijk aan (FUNCALL f_1 arg_1 · · ·).

Bijvoorbeeld,

```
(FUNCALL (COMPOSITION 'CAR 'CDR) '(A B C))
```

is hetzelfde als

```
(FUNCALL 'CAR (FUNCALL 'CDR '(A B C)))
```

2. (INSERT functie) past een functie toe op een reeks argumenten en accumuleert het resultaat.

```
(FUNCALL (INSERT functie) lijst)
```

met lijst een reeks argumenten en functie een functie, is gelijk aan

```
(FUNCALL functie
  (CAR lijst)
  (FUNCALL (INSERT functie) (CDR lijst)))
```

als lijst meer dan één element bevat en gelijk aan (CAR lijst) als (CDR lijst) NIL is.

Bijvoorbeeld:

```
(FUNCALL (INSERT '+) '(1 5 2))

⇒ (FUNCALL '+ 1 (FUNCALL (INSERT '+) '(5 2)))
⇒ (FUNCALL '+ 1 (FUNCALL '+ 5
                         (FUNCALL (INSERT '+ '(2)))))
⇒ (FUNCALL '+ 1 (FUNCALL '+ 5 2))
⇒ (FUNCALL '+ 1 7)
⇒ 8
```

3. (APPLY-TO-ALL functie) past een functie toe op elk element, zodanig dat

   (FUNCALL (APPLY-TO-ALL functie) '(arg_1 · · · arg_n))

   gelijk is aan

   (LIST (FUNCALL functie 'arg_1) · · · (FUNCALL functie 'arg_n))

   Bijvoorbeeld (FUNCALL (APPLY-TO-ALL '1+) '(1 2 3)) is gelijk aan (2 3 4).

Het merkwaardige van een functionele stijl is dat de argumenten volledig verdwijnen uit de definitie. Er wordt een manipulatie van functies beschreven en het resultaat wordt toegepast op de argumenten. De functie LENGTH die het aantal elementen in een lijst berekent is bijvoorbeeld gelijk aan

```
(COMPOSITION
  (INSERT '+)
  (APPLY-TO-ALL (CONSTANTE 1)))
```

Als we het symbool F-LENGTH binden aan deze uitdrukking:

```
(SETQ F-LENGTH
  (COMPOSITION
    (INSERT '+)
    (APPLY-TO-ALL (CONSTANTE 1))))
```

dan is

```
(FUNCALL F-LENGTH '(A B C))
```

gelijk aan 3.

Een functionele stijl lijkt veel minder efficiënt dan een applicatieve (of imperatieve) stijl. Dit is echter niet noodzakelijk zo. Als een array-processor wordt gebruikt, of parallellisme, of een speciale functionele architectuur, kan een functioneel geschreven programma zelfs sneller werken. Anderzijds kan een slimme compilator altijd de mogelijke optimaliseringen uitvoeren.

Een functionele programmeerstijl leidt voor sommige problemen tot erg compacte programma's die iets gemeen hebben met haiku's wat betreft verrassende gevatheid. Bijvoorbeeld LAST ziet er functioneel als volgt uit:

```
(INSERT 'TWEE)
```

waarbij TWEE een functie is die zijn tweede argument weergeeft:

```
(DEFUN TWEE (X Y) Y)
```

Er zijn programmeertalen die een functionele programmeerstijl opleggen. Het bekendste voorbeeld hiervan is FP (Backus, 1978). Zoals de voorbeelden uit dit hoofdstuk aantonen kan men echter evengoed een functionele stijl in LISP gebruiken, zonder andere stijlen die meer geschikt zijn voor andere soorten van problemen, uit te sluiten.

5.2. OEFENINGEN

1. Schets de evaluatie van (FUNCALL F-LENGTH '(A B C))

(FUNCALL F-LENGTH '(A B C)) is gelijk aan

```
(FUNCALL
  (COMPOSITION
    (INSERT '+)
    (APPLY-TO-ALL (CONSTANTE 1)))
  '(A B C))
```

en dus

```
(FUNCALL
  (INSERT '+)
  (FUNCALL
    (APPLY-TO-ALL (CONSTANTE 1))
    '(A B C)))

(FUNCALL
  (APPLY-TO-ALL (CONSTANTE 1))
  '(A B C))
```

is gelijk aan

```
(LIST
  (FUNCALL (CONSTANTE 1) 'A)
  (FUNCALL (CONSTANTE 1) 'B)
  (FUNCALL (CONSTANTE 1) 'C))
```

en (CONSTANTE 1) is gelijk aan (LAMBDA (Y) 1), zodanig dat deze uitdrukking gelijk is aan (1 1 1).

```
(FUNCALL
  (INSERT '+) '(1 1 1))
```

is gelijk aan

```
(FUNCALL '+ 1
  (FUNCALL '+ 1 1))
```

of 3.

2. Ga aan de hand van voorbeelden na dat LAST gelijk is aan (INSERT 'TWEE).

We bekijken de berekening van het laatste element van '(1 2 3). (FUNCALL (INSERT 'TWEE) '(3)) is gelijk aan 3 omdat L slechts één argument heeft. (FUNCALL (INSERT 'TWEE) '(2 3)) is gelijk aan

```
(FUNCALL 'TWEE
  2
  (FUNCALL (INSERT 'TWEE) '(3)))
```

of

```
(FUNCALL 'TWEE 2 3)
```

wat gelijk is aan 3. Tenslotte

```
(FUNCALL (INSERT 'TWEE) '(1 2 3))
```

is gelijk aan

```
(FUNCALL
  F
  (CAR L)
  (APPLY (INSERT F) (CDR L)))
```

of

```
(FUNCALL 'TWEE 1 3)
```

wat gelijk is aan 3.

3. Schrijf een hulpfunctie DISTL (distribute left) die gegeven een element en een lijst van elementen, paren construeert. Bijvoorbeeld (DISTL 'A '(A B C)) is gelijk aan ((A A) (A B) (A C)).

Dit is een eenvoudige MAP-functie:

```
(DEFUN DISTL (ELEMENT LIJST)
  (MAPCAR
    #'(LAMBDA (X) (LIST ELEMENT X))
    LIJST))
```

4. Definieer de functie MEMBER in een functionele stijl.

MEMBER brengt eerst de argumenten in de juiste volgorde met DISTL. Dan past ze EQ toe op de bekomen paren, en combineert het resultaat met OR:

```
(COMPOSITION
  (INSERT 'OR)
  (APPLY-TO-ALL 'EQ)
  'DISTL)
```

5. Definieer de functie APPEND in functionele stijl.

```
(INSERT 'CONS)
```

6. Definieer een functie die het inwendig produkt van twee vectoren berekent in functionele stijl. Het inwendig produkt is gelijk aan

$$\sum_{i=1}^{n} u_i v_i$$

De vectoren worden voorgesteld als lijsten. Bijvoorbeeld het inwendig produkt van (1 2 3) en (6 5 4) is

$$(1 \times 6) + (2 \times 5) + (3 \times 4) = 28.$$

INWENDIG-PRODUKT is gelijk aan:

```
(COMPOSITION
  (INSERT '+)
  (APPLY-TO-ALL '*)
  TRANSPOSE))
```

TRANSPOSE dient om de argumenten op de juiste plaats te brengen:

```
(DEFUN TRANSPOSE (VECTOR-A VECTOR-B)
  (MAPCAR
    #'(LAMBDA (X Y) (LIST X Y))
    VECTOR-A VECTOR-B))
```

(TRANSPOSE '(1 2 3) '(6 5 4)) is gelijk aan ((1 6) (2 5) (3 4)). (FUNCALL (APPLY-TO-ALL '*) '((1 6) (2 5) (3 4))) geeft (6 10 12) en de som hiervan is 28.

7. Definieer de functie APPLY-TO-ALL.

APPLY-TO-ALL moet een lambda-uitdrukking opleveren die het gewenste resultaat heeft. De lambda-uitdrukking kan een MAPCAR gebruiken om de iteratie uit te voeren. Bijvoorbeeld voor (FUNCALL (APPLY-TO-ALL '1+) '(1 2 3)) is dit

```
(FUNCALL
  #'(LAMBDA (ARG)
      (MAPCAR #'(LAMBDA (Y) (1+ Y)) ARG))
  '(1 2 3)).
```

De definitie ziet er zo uit:

```
(DEFUN APPLY-TO-ALL (FUNCTIE)
  #'(LAMBDA (LIJST)
      (MAPCAR
        #'(LAMBDA (ELEMENT)
            (FUNCALL FUNCTIE ELEMENT))
        LIJST)))
```

8. Definieer de functie INSERT.

INSERT moet opnieuw een lambda-uitdrukking opleveren. Laten we een hulpfunctie INSERT2 gebruiken die de nodige evaluaties uitvoert, zodanig dat

```
(FUNCALL (INSERT '+) ARG)
```

gelijk is aan

```
(FUNCALL #'(LAMBDA (L) (INSERT2 + L)) ARG).
```

INSERT wordt als volgt gedefinieerd:

```
(DEFUN INSERT (FUNCTIE)
  #'(LAMBDA (LIJST) (INSERT2 FUNCTIE LIJST)))
```

waarbij

```
(DEFUN INSERT2 (FUNCTIE ARGLIJST)
 (COND
  ((NULL (CDR ARGLIJST))
    ;; als er slechts een element is, is dit het resultaat
    (CAR ARGLIJST))
  (T
    ;; anders, pas de functie toe
    (FUNCALL FUNCTIE
      (CAR ARGLIJST)
      (INSERT2 FUNCTIE (CDR ARGLIJST))))))
```

6. SAMENVATTING

In dit deel hebben we bestudeerd hoe functies als volwaardige objecten in LISP kunnen gebruikt worden: het is mogelijk een functie als argument te hebben van een andere functie en een functie dynamisch samen te stellen.

1. (APPLY F A) past de functie F toe op de lijst van argumenten A.
2. (FUNCALL F $A_1 \cdots A_n$) past de functie F toe op de evaluatie van $A_1 \cdots A_n$.
3. (MAPCAR F $A_1 \cdots A_n$) past de functie F toe op successieve elementen in elk van de argumenten en combineert het resultaat met CONS.
4. (MAPCAN F $A_1 \cdots A_n$) past de functie F toe op successieve elementen in elk van de argumenten en combineert het resultaat met APPEND.
5. (LAMBDA ($A_1 \cdots A_n$) $formule_1 \cdots formule_n$) is gelijk aan een functie die de argumenten $A_1 \cdots A_n$ bindt en daarna de formules met deze bindingen uitvoert.

Het gebruik van deze functies van hogere orde leidt tot een stijl van programmeren die noch recursief, noch imperatief is. Het is zelfs mogelijk om programma's te schrijven die alleen maar gebruik maken van functies van hogere orde en bestaan uit operaties over functies. Deze stijl van programmeren heet functioneel

programmeren.

7. GEMENGDE OPGAVEN

Net zoals in het vorige deel bekijken we nu enkele problemen van grotere omvang.

7.1. DE OPGAVEN

7.1.1. LINEAIRE REGRESSIE

Lineaire regressie is een methode om een lijn

$$y = ax + b$$

te vinden die best past bij een reeks data $(x_i, y_i)_{i=1..n}$. De coefficiënten a en b volgen uit de volgende formules:

$$a = \frac{\sum_{i=1}^{n} y_i - b(\sum_{i=1}^{n} x_i)}{n}$$

en

$$b = \frac{n(\sum_{i=1}^{n} x_i y_i) - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n(\sum_{i=1}^{n} x_i^2) - (\sum_{i=1}^{n} x_i)^2}$$

Schrijf een functie die a en b berekent voor een reeks van paren x en y. Gebruik zoveel mogelijk MAPCAR.

7.1.2. MATRIX ALGEBRA

Schrijf enkele belangrijke functies uit de matrix-algebra:

(GELIJKE-MATRICES M1 M2) gaat na of twee matrices gelijk zijn.

(GELIJKE-DIMENSIES M1 M2) gaat of twee matrices gelijke dimensies hebben.

(MATRIX-SOM M1 M2) geeft de som van twee matrices.

(MATRIX-VERSCHIL M1 M2) geeft het verschil van twee matrices.

(MATRIX-SCALAIR-PRODUKT SCALAIR MATRIX) geeft het scalair produkt van een matrix en een getal.

(MATRIX-PRODUKT M1 M2) geeft het produkt van twee matrices.

Veronderstel dat de matrices gerepresenteerd worden met lijsten: een deellijst voor elke rij. Bijvoorbeeld:

((4 3 1) (2 0 8) (6 8 9))

stelt de volgende matrix voor:

$$\begin{pmatrix} 4 & 3 & 1 \\ 2 & 0 & 8 \\ 6 & 8 & 9 \end{pmatrix}$$

7.2. OPLOSSINGEN

7.2.1. LINEAIRE REGRESSIE

Dit is een eenvoudige toepassing van de programmeermechanismen die we in dit deel hebben gezien. We berekenen eerst een lijst van x-waarden en een lijst van y-waarden, dan de diverse sommen, dan b en tenslotte a:

```
(DEFUN LINEAIRE-REGRESSIE (X-Y-REEKS)
  (LET
   ((N (F-LENGTH DATA))   ; aantal elementen
    (LIJST-X (MAPCAR 'CAR X-Y-REEKS))   ; lijst van x-waarden
    (LIJST-Y (MAPCAR 'CADR X-Y-REEKS)))   ; lijst van y-waarden
   (LET
    ((SUM-X (APPLY '+ LIJST-X))   ; som van x-waarden
     (SUM-Y (APPLY '+ LIJST-Y))   ; som van y-waarden
     (SUM-2-X        ; som van x^2
       (APPLY '+
         (MAPCAR
           #'(LAMBDA (X)
               (EXP X 2))
           LIJST-X))))
     (SUM-X-Y      ;som van produkten van x en y waarden
       (APPLY '+
         (MAPCAR
           #'(LAMBDA (X Y)
               (* X Y))
           LIJST-X LIJST-Y))))
    (LET
     ((B   ; berekening van B
       (/
        (-
         (* N SUM-X-Y)
         (* SUM-X SUM-Y))
        (-
         (* N SUM-2-X)
         (EXPT SUM-X 2)))))
      (LIST
       ;; berekening van A aan de hand van B
       (/
        (- SUM-Y
           (* B SUM-X))
        N)
       B))))
```

7.2.2. MATRIX ALGEBRA

Twee matrices zijn aan elkaar gelijk als ze dezelfde elementen hebben op dezelfde plaats. In termen van lijsten betekent dit dat de lijsten gelijk moeten zijn:

```
(DEFUN GELIJK-MATRIX (M1 M2)
  (EQUAL M1 M2))
```

Twee matrices hebben gelijke dimensies als ze hetzelfde aantal rijen en hetzelfde aantal kolommen hebben.

```
(DEFUN GELIJKE-DIMENSIES (M1 M2)
  (AND
    ;; is de lengte van de lijsten in hun geheel gelijk
    (= (F-LENGTH M1) (F-LENGTH M2))
    ;; is de lengte van elke deellijst gelijk?
    (= (F-LENGTH (CAR M1)) (F-LENGTH (CAR M2)))))
```

Twee matrices kunnen slechts opgeteld worden als ze gelijke dimensies hebben. Het optellen is een eenvoudige mapping operatie:

```
(DEFUN MATRIX-+ (M1 M2)
  (COND
    ((NOT (GELIJKE-DIMENSIES M1 M2)) 'UNDEFINED)
    (T
      (MAPCAR
        '(LAMBDA (RIJ1 RIJ2)
          (MAPCAR '(LAMBDA (X Y) (+ X Y))
                  RIJ1 RIJ2))
        M1 M2))))
```

Hetzelfde geldt voor MATRIX-VERSCHIL:

```
(DEFUN MATRIX-VERSCHIL (M1 M2)
  (COND
    ((NOT (GELIJKE-DIMENSIES M1 M2)) 'UNDEFINED)
    (T
     (MAPCAR
       '(LAMBDA (RIJ1 RIJ2)
          (MAPCAR '(LAMBDA (X Y) (- X Y))
                  RIJ1 RIJ2))
       M1 M2))))
```

Het matrix scalair produkt is gelijk aan het produkt van elk element in de matrix met het scalair getal:

```
(DEFUN MATRIX-SCALAIR-PRODUKT (SCALAIR MATRIX)
  (MAPCAR
    '(LAMBDA (RIJ)
      (MAPCAR
        '(LAMBDA (ELEMENT)
          (* SCALAIR ELEMENT))
        RIJ))
    MATRIX))
```

Het produkt van twee matrices M1 en M2 kan enkel berekend worden als M1 evenveel kolommen heeft als M2 rijen. Hier is een functie die deze conditie nagaat:

```
(DEFUN EVENVEEL-RIJEN-KOLOMMEN (M1 M2)
  (= (F-LENGTH (CAR M1)) (F-LENGTH M2)))
```

Om het produkt te berekenen zetten we eerst de tweede matrix om zodat kolommen rijen worden en rijen kolommen. Dit kan met de volgende functie:

```
(DEFUN ZET-MATRIX-OM (M)
  (COND
    ((NULL (CAR M)) NIL)
    (T
      (CONS
        (MAPCAR '(LAMBDA (X) (CAR X)) M)
        (ZET-MATRIX-OM
          (MAPCAR '(LAMBDA (X) (CDR X)) M))))))
```

Dan voeren we de nodige mapping-operaties uit:

```
(DEFUN MATRIX-PRODUKT (M1 M2)
  (COND
    ((NOT (EVENVEEL-RIJEN-KOLOMMEN M1 M2))
      'UNDEFINED)
    (T
      (LET
        ((OMZETTING-M2 (ZET-MATRIX-OM M2)))
        (MAPCAR
          '(LAMBDA (RIJ)
            (MAPCAR
              '(LAMBDA (KOLOM)
                (APPLY
                  '+
                  (MAPCAR
                    '(LAMBDA
                       (RIJ-ELEMENT KOLOM-ELEMENT)
                      (* RIJ-ELEMENT KOLOM-ELEMENT))
                    RIJ KOLOM)))
              OMZETTING-M2))
          M1)))))
```

7.3. GEMENGDE OPGAVEN

1. Definieer APPLY met FUNCALL.
2. Definieer de functie SORT.
3. Definieer de combinator COMPOSITION.

DEEL 5. STRUCTUREN

Dit hoofdstuk gaat over structuren. We bekijken hoe abstracte datastructuren kunnen gerepresenteerd worden in LISP en hoe macro's de inefficiënties van deze vorm van abstractie kunnen wegwerken. We bekijken ook twee primitieve datastructuren van LISP: eigenschaplijsten en arrays. Vervolgens bekijken we hoe we in LISP ook objectgericht kunnen programmeren. Het laatste hoofdstuk behandelt enkele technieken om grote programma's in LISP gemakkelijk te kunnen ontwikkelen en onderhouden.

1. ABSTRACTE DATASTRUCTUREN

1.1. DEFINITIES

Een *datastructuur* is een representatie van een datatype en een reeks van functies die opereren over deze representatie. Een typisch voorbeeld is de *lijst* met functies:

1. om elementen te construeren, de *constructors* (bijv. CONS en APPEND).
2. om gedeelten van elementen te adresseren, de *selectors* (bijv. CAR en CDR).
3. om bepaalde eigenschappen van elementen na te gaan, de *predikaten* (bijv. NULL of MEMBER).

Een *abstracte datastructuur* (ook type genoemd) is een datastructuur die door de programmeur zelf is gedefinieerd. De definitie van een abstracte datastructuur omvat de definitie van een interne representatie in termen van andere datatypen en datastructuren, en de definitie van constructors, selectors en predikaten in termen van operaties over deze andere datatypen en datastructuren. Een verzameling kan bijvoorbeeld gerepresenteerd worden als een lijst. Functies zoals VOEG-ELEMENT-TOE, IS-LID? en LEGE-VERZAMELING? zijn dan gedefinieerd in termen van functies over lijsten.

Het voordeel van abstracte datastructuren is dat ze programma's overzichtelijker maken en dus minder complex. Een functie CARDINALITEIT toegepast op een element van het type verzameling is duidelijker dan een functie LENGTH, alhoewel intern CARDINALITEIT misschien volledig gelijk is aan LENGTH. Abstracte datastructuren maken programma's ook meer modulair: iemand kan de interne representatie of een datastructuurfunctie wijzigen zonder dat hij de programma's die deze datastructuur gebruiken, moet veranderen. Ook kan een andere programmeur het pakket van functies dat de datastructuur definieert in een ander programma gebruiken.

LISP is uitzonderlijk goed geschikt om abstracte datastructuren te definiëren omdat er geen restricties zijn op het type van de argumenten in een functie, omdat de

inefficiënties van abstracte datastructuren kunnen worden weggecompileerd en omdat lijsten krachtige primitieve datastructuren zijn waaruit andere structuren gemakkelijk kunnen worden opgebouwd.

Bij wijze van voorbeeld bekijken we nu in de oefeningen een abstracte datastructuur voor verzamelingen. Een verzameling is intern gelijk aan een lijst, de lege verzameling aan de lege lijst.

1.2. OEFENINGEN

1. Schrijf een predikaat LEGE-VERZAMELING? dat nagaat of een verzameling de lege verzameling is

---

```
(DEFUN LEGE-VERZAMELING? (VERZAMELING)
  (NULL VERZAMELING))
```

2. Schrijf een functie die de cardinaliteit van een verzameling berekent.

```
(DEFUN CARDINALITEIT (VERZAMELING)
  (LENGTH VERZAMELING))
```

3. Schrijf een functie EERSTE-ELEMENT die het eerste element van een verzameling oplevert.

```
(DEFUN EERSTE-ELEMENT (VERZAMELING)
  (CAR VERZAMELING))
```

4. Schrijf een functie RESTERENDE-ELEMENTEN die de elementen van een verzameling behalve het eerste oplevert.

```
(DEFUN RESTERENDE-ELEMENTEN (VERZAMELING)
  (CDR VERZAMELING))
```

5. Schrijf een predikaat IS-LID? om na te gaan of een element lid is van een verzameling.

```
(DEFUN IS-LID? (ELEMENT VERZAMELING)
  (MEMBER ELEMENT VERZAMELING))
```

6. Schrijf een predikaat GELIJKE-VERZAMELINGEN? dat nagaat of twee verzamelingen gelijk zijn.

We schrijven eerst een hulp-predikaat DEELVERZAMELING?, met als argumenten twee verzamelingen, dat nagaat of de eerste verzameling een deelverzameling van de tweede is. Omdat de orde van de elementen in een verzameling geen rol speelt kunnen we niet EQUAL gebruiken maar moeten we nagaan of elk element van de ene verzameling in de andere verzameling voorkomt.

```
(DEFUN DEELVERZAMELING?
          (VERZAMELING-1 VERZAMELING-2)
  (COND
    ((LEGE-VERZAMELING? VERZAMELING-1)
      T)
    ((IS-LID?
       (EERSTE-ELEMENT VERZAMELING-1)
       VERZAMELING-2)
      (DEELVERZAMELING?
        (RESTERENDE-ELEMENTEN VERZAMELING-1)
        VERZAMELING-2))
    (T NIL)))
```

Nu kunnen we gemakkelijk nagaan of twee verzamelingen gelijk zijn: het volstaat dat elk een deelverzameling van de andere is.

```
(DEFUN GELIJKE-VERZAMELINGEN?
          (VERZAMELING-1 VERZAMELING-2)
  (AND
    (DEELVERZAMELING? VERZAMELING-1 VERZAMELING-2)
    (DEELVERZAMELING? VERZAMELING-2 VERZAMELING-1)))
```

7. Herschrijf nu IS-LID? zodanig dat de elementen zelf ook verzamelingen kunnen zijn.

```
(DEFUN IS-LID? (ELEMENT VERZAMELING)
  (COND
    ((ATOM ELEMENT) (MEMBER ELEMENT VERZAMELING))
    (T (IS-LID-2 ELEMENT VERZAMELING))))
```

waarbij IS-LID-2 een recursieve hulpfunctie is die nagaat of een verzameling lid is van een verzameling:

```
(DEFUN IS-LID-2 (ELEMENT VERZAMELING)
  (COND
    ((LEGE-VERZAMELING? VERZAMELING) NIL)
    ((GELIJKE-VERZAMELINGEN?
       ELEMENT
       (EERSTE-ELEMENT VERZAMELING))
      T)
    (T
      (IS-LID-2
        ELEMENT
        (RESTERENDE-ELEMENTEN VERZAMELING)))))
```

8. Schrijf een functie VOEG-ELEMENT-TOE om een element toe te voegen aan een verzameling.

```
(DEFUN VOEG-ELEMENT-TOE (ELEMENT VERZAMELING)
  (COND
    ((IS-LID? ELEMENT VERZAMELING)
      VERZAMELING)
    (T
      (CONS ELEMENT VERZAMELING))))
```

We moeten immers nagaan of het element al aanwezig is omdat gelijke elementen in een verzameling slechts als één element gelden.

Het is ondertussen wel duidelijk hoe we geleidelijk een vocabularium van functies en predikaten opbouwen om met verzamelingen te werken. Op die manier wordt verzameling een datastructuur zoals lijst of getal en een andere gebruiker hoeft niet te weten hoe verzamelingen intern zijn gerepresenteerd of hoe de functies voor deze datastructuur werken. We bekijken nu een tweede voorbeeld van een datastructuur, de stapel, en exploreren nu hoe macro's de

extrakost van abstracte datastructuren kunnen wegwerken.

9. Een stapel is een datastructuur waarbij elementen alleen bij het begin of het einde toegevoegd en afgevoerd kunnen worden. Veronderstel dat stapels als lijsten zijn gerepresenteerd en definieer functies voor PUSH (toevoegen van een element aan een stapel), POP (afvoeren van een element), EERSTE-ELEMENT (geeft het eerste element van een stapel), LEGE-STAPEL? (gaat na of een stapel leeg is), en INIT-STAPEL (initialiseert een stapel). Bijvoorbeeld, de evaluatie van

```
(LET*
  ((S (INIT-STAPEL))
   (S (PUSH 5 S)))
  (PRINT (EERSTE-ELEMENT S))
  (LET*
    ((S (PUSH 10 S))
     (S (POP S))
    (PRINT (EERSTE-ELEMENT S)))))
```

brengt eerst 5 en dan terug 5 op het scherm. (LET* is de sequentiële versie van LET). Gebruik zoveel mogelijk macro's.

```
(DEFMACRO INIT-STAPEL ()
  NIL)

(DEFMACRO PUSH (ELEMENT STAPEL)
  '(CONS ,ELEMENT ,STAPEL))

(DEFMACRO POP (STAPEL)
  '(CDR ,STAPEL))

(DEFMACRO EERSTE-ELEMENT (STAPEL)
  '(CAR ,STAPEL))

(DEFMACRO LEGE-STAPEL? (STAPEL)
  '(NULL ,STAPEL))
```

10. Wat is de expansie van

```
(LET*
  ((S (INIT-STAPEL))
   (S (PUSH 5 S)))
  (PRINT (EERSTE-ELEMENT S))
  (LET*
    ((S (PUSH 10 S))
     (S (POP S))
    (PRINT (EERSTE-ELEMENT S)))))
```

```
(LET*
  ((S NIL)
   (S (CONS 5 S)))
  (PRINT (CAR S))
  (LET*
    ((S (CONS 10 S))
     (S (CDR S))
    (PRINT (CAR S)))))
```

Het is duidelijk dat de versie met de abstracte functies heel wat overzichtelijker is dan de versie zonder deze functies.

11. Stapels kunnen ook in een imperatieve stijl worden geïmplementeerd. Functies zoals PUSH, POP, enz. veranderen dan de toestand van de stapel via een neveneffect. Bijvoorbeeld

```
(PROG ()
  (INIT-STAPEL 'S)
  (PUSH 5 'S)
  (PRINT (EERSTE-ELEMENT 'S))
  (PUSH 10 'S)
  (POP 'S)
  (PRINT (EERSTE-ELEMENT 'S)))
```

brengt eerst 5 en dan terug 5 op het scherm. Herschrijf de stapelfuncties in deze zin.

De elementen van een stapel zijn gelijk aan de waarde gebonden aan de stapel.

```
(DEFMACRO INIT-STAPEL (S)
  `(SET ,S NIL))
```

```
(DEFMACRO PUSH (ELEMENT STAPEL)
  '(SET ,STAPEL (CONS ,ELEMENT (EVAL ,STAPEL))))

(DEFMACRO POP (STAPEL)
  '(SET ,STAPEL (CDR (EVAL ,STAPEL))))

(DEFMACRO EERSTE-ELEMENT (STAPEL)
  '(CAR (EVAL ,STAPEL)))

(DEFMACRO LEGE-STAPEL? (STAPEL)
  '(NULL (EVAL ,STAPEL)))
```

12. Geef de expansie van

```
(PROG ()
  (INIT-STAPEL 'S)
  (PUSH 5 'S)
  (PRINT (EERSTE-ELEMENT 'S))
  (PUSH 10 'S)
  (POP 'S)
  (PRINT (EERSTE-ELEMENT 'S)))
```

volgens deze nieuwe definities.

```
(PROG (S)
  (SET 'S NIL)
  (SET 'S (CONS 5 (EVAL 'S)))
  (PRINT (CAR (EVAL 'S)))
  (SET 'S (CONS 10 (EVAL 'S)))
  (SET 'S (CDR (EVAL 'S)))
  (PRINT (CAR (EVAL 'S))))
```

2. EIGENSCHAPLIJSTEN

2.1. DEFINITIES

De eigenschaplijst is een primitieve LISP-datastructuur die dient om een reeks eigenschappen te associëren met een atoom. De eigenschaplijst is een belangrijke primitieve datastructuur om andere datastructuren te maken, vooral datastructuren die veranderlijke informatie bijhouden.

Een eigenschap op een eigenschaplijst heeft twee componenten. De naam van de eigenschap en de waarde. Bijvoorbeeld LEEFTIJD is de naam van een eigenschap en 15 een mogelijke waarde.

1. De functie GET zoekt de waarde van een eigenschap op voor een atoom. Bijvoorbeeld (GET 'JAN 'LEEFTIJD) levert de waarde geassocieerd met LEEFTIJD op de eigenschaplijst van JAN. GET is gelijk aan NIL als de eigenschap niet aanwezig is.
2. De functie PUTPROP voegt een waarde voor een eigenschap toe aan een eigenschaplijst. Bijvoorbeeld (PUTPROP 'JAN 15 'LEEFTIJD) associeert de waarde 15 met LEEFTIJD op de eigenschaplijst van JAN. Als er reeds een waarde aanwezig is, wordt die verwijderd.
3. De functie REMPROP ("remove property") verwijdert een eigenschap van de eigenschaplijst van een atoom. Bijvoorbeeld (REMPROP 'JAN 'LEEFTIJD) verwijdert LEEFTIJD en zijn waarde van de eigenschaplijst van JAN.
4. De functie PLIST ("property list") levert de eigenschaplijst van het atoom X.

Intern is een eigenschaplijst gelijk aan een lijst van "dotted pairs". Een dotted pair is een cons waarvan de cdr niet gelijk is aan NIL maar aan een atoom. Het wordt opgeschreven met een punt. Bijvoorbeeld (A . B) is een dotted pair. De car van deze lijst is A en de cdr van deze lijst is het atoom B. (A . B) kan bekomen worden door (CONS 'A 'B).

De eigenschaplijst is het primitieve bouwelement voor het construeren van een gegevensbestand en wordt ook door het LISP systeem zelf gebruikt om interne informatie te bewaren. De functies PUTPROP en REMPROP hebben duidelijk een imperatief karakter: ze worden geëvalueerd omwille van een neveneffect. Het neveneffect is een verandering in de datastructuren.

2.2. OEFENINGEN

1. Associeer de kleur blauw met het atoom lucht.

(PUTPROP 'LUCHT 'BLAUW 'KLEUR)

2. Verander de kleur in grijs.

(PUTPROP 'LUCHT 'GRIJS 'KLEUR)

3. Hoe kom je de kleur van lucht te weten?

Met (GET 'LUCHT 'KLEUR) dat GRIJS oplevert.

4. Verwijder de eigenschap KLEUR van LUCHT.

(REMPROP 'LUCHT 'KLEUR)

5. Implementeer de stapelfuncties aan de hand van eigenschaplijsten. Elke stapel heeft een eigenschap ELEMENTEN die de elementen van de stapel bevat.

```
(DEFMACRO INIT-STAPEL (S)
  `(PUTPROP ,S NIL 'ELEMENTEN))

(DEFMACRO PUSH (ELEMENT STAPEL)
  `(PUTPROP ,STAPEL
            (CONS ,ELEMENT (GET ,STAPEL 'ELEMENTEN))
            'ELEMENTEN))

(DEFMACRO POP (STAPEL)
  `(PUTPROP ,STAPEL
            (CDR ,(GET ,STAPEL 'ELEMENTEN))
            'ELEMENTEN))

(DEFMACRO EERSTE-ELEMENT (STAPEL)
  `(CAR (GET ,STAPEL 'ELEMENTEN)))

(DEFMACRO LEGE-STAPEL? (STAPEL)
  `(NULL (GET ,STAPEL 'ELEMENTEN)))
```

6. Geef de expansie van

```
(PROG ()
  (INIT-STAPEL 'S)
  (PUSH 5 'S)
  (PRINT (EERSTE-ELEMENT 'S))
  (PUSH 10 'S)
  (POP 'S)
  (PRINT (EERSTE-ELEMENT 'S)))
```

volgens deze nieuwe definities.

```
(PROG ()
  (PUTPROP 'S NIL 'ELEMENTEN)
  (PUTPROP 'S (CONS 5 (GET 'S 'ELEMENTEN) 'ELEMENTEN))
  (PRINT (CAR (GET 'S 'ELEMENTEN)))
  (PUTPROP 'S
           (CONS 10 (GET 'S 'ELEMENTEN))
           'ELEMENTEN)
  (PUTPROP 'S
           (CDR (GET 'S 'ELEMENTEN))
           'ELEMENTEN)
  (PRINT (CAR (GET 'S 'ELEMENTEN))))
```

Merk op hoe het mogelijk was de implementatie van de stapelfuncties te veranderen zonder de PROG die deze functies gebruikt te wijzigen.

3. ARRAYS

3.1. DEFINITIES

Een array is een compacte manier om een reeks objecten te groeperen en als eenheid te behandelen. Elk element zit in een bepaalde cel die via een getal (een index) expliciet kan worden geadresseerd. Een array heeft dimensies die aangeven hoeveel indices er nodig zijn om naar een element te refereren. Een array heeft ook een type dat aangeeft wat het type is van de elementen in de array. Fixnum en flonum zijn mogelijke typen. In de meeste implementaties kan gelijk welk object in een arraycel worden gestopt. MAKE-ARRAY, ASET, AREF en ARRAY-DIMENSIONS zijn de belangrijkste functies voor arrays.

De functie MAKE-ARRAY construeert een array. Het heeft een of meer argumenten. Het eerste argument geeft de extensie van elke dimensie aan: een lijst met getallen. Alle andere argumenten zijn &KEY argumenten waarmee verdere opties (zoals bijvoorbeeld het type van de array) kunnen gespecifieerd worden. Bijvoorbeeld,

```
(MAKE-ARRAY '(2 2) :ELEMENT-TYPE 'SINGLE-FLOAT)
```

levert een array op van 2 rijen met elk 2 kolommen. Elk element van de array kan een single precision floating point getal bevatten.

De functie ASET plaatst een element in een cel van een array. ASET heeft drie of meer argumenten, het eerste argument is een formule waarvan de waarde moet worden opgeslagen. Het tweede element is de naam van de array en de overige elementen beschrijven de positie van de cel. Bijvoorbeeld, de volgende uitdrukking

```
(ASET (+ 1 5) FOO 1 1)
```

plaatst 6 in de 2de cel van de 2de rij van FOO.

AREF dient om een element uit een cel van een array te adresseren. Het eerste argument van AREF is de naam van de array. De overige argumenten beschrijven de positie van de cel. Bijvoorbeeld

```
(AREF FOO 1 1)
```

geeft het element dat in de 2de cel van de 2de rij van FOO zit.

(ARRAY-DIMENSIONS X) geeft de dimensies van X in de vorm van een lijst. Bijvoorbeeld gegeven

```
(SETQ FOO (MAKE-ARRAY '(2 2)))
```

dan is (ARRAY-DIMENSIONS FOO) gelijk aan de lijst (2 2).

3.2. OEFENINGEN

1. We gaan de functies voor matrices uit de gemengde opgaven van het vorige deel herschrijven met een array als basisrepresentatie. Een matrix is een array met n rijen en m kolommen. Schrijf eerst een functie MAAK-MATRIX die gegeven m en n een matrix construeert.

```
(DEFMACRO MAAK-MATRIX (M N)
  '(MAKE-ARRAY '(,M ,N)))
```

2. Herschrijf de functie (GELIJKE-DIMENSIES M1 M2).

```
(DEFMACRO GELIJKE-DIMENSIES (M1 M2)
  '(EQUAL
     (ARRAY-DIMENSIONS ,M1)
     (ARRAY-DIMENSIONS ,M2)))
```

3. Herschrijf de functie MATRIX-PLUS die de som van twee matrices berekent. Gebruik DO.

```
(DEFUN MATRIX-PLUS (M1 M2)
  ;; zoek eerst de dimensies op van M1 en M2
  (LET
    ((DIMS-M1 (ARRAY-DIMENSIONS M1))
     (DIMS-M2 (ARRAY-DIMENSIONS M2)))
    (COND
      ;; als de dimensies niet gelijk zijn is er geen som
      ((NOT (EQUAL DIMS-M1 DIMS-2)) 'UNDEFINED)
      (T
       ;; anders maak een nieuwe matrix
       (LET*
         ((AANTAL-RIJEN (CAR DIMS-M1))
          (AANTAL-KOLOMMEN (CADR DIMS-M1))
          (NIEUWE-MATRIX
         (MAAK-MATRIX AANTAL-RIJEN AANTAL-KOLOMMEN)))
         ;; I en J lopen door de rijen en kolommen
         (DO
           ((I 1 (1+ I)))
           ((= I AANTAL-RIJEN) NIEUWE-MATRIX)
           (DO
             ((J 1 (1+ J)))
             ((= J AANTAL-KOLOMMEN) T)
             ;; de som wordt opgeslagen in de nieuwe matrix
             (ASET
               (+ (AREF M1 I J) (AREF M2 I J))
               NIEUWE-MATRIX I J))))))))
```

4. Herschrijf de functie MATRIX-VERSCHIL die het verschil van twee matrices berekent.

```
(DEFUN MATRIX-VERSCHIL (M1 M2)
  ;; zoek eerst de dimensies op van M1 en M2
  (LET
   ((DIMS-M1 (ARRAY-DIMENSIONS M1))
    (DIMS-M2 (ARRAY-DIMENSIONS M2)))
   (COND
     ;; als de dimensies niet gelijk zijn is er geen som
     ((NOT (EQUAL DIMS-M1 DIMS-2)) 'UNDEFINED)
     (T
      ;; anders maak een nieuwe matrix
      (LET*
        ((AANTAL-RIJEN (CAR DIMS-M1))
         (AANTAL-KOLOMMEN (CADR DIMS-M1))
         (NIEUWE-MATRIX
           (MAAK-MATRIX AANTAL-RIJEN AANTAL-KOLOMMEN)))
        ;; I en J lopen door de rijen en kolommen
        (DO
          ((I 1 (1+ I)))
          ((= I AANTAL-RIJEN) NIEUWE-MATRIX)
          (DO
            ((J 1 (1+ J)))
            ((= J AANTAL-KOLOMMEN) T)
            ;; het verschil wordt opgeslagen in de nieuwe matrix
            (ASET
              (- (AREF M1 I J) (AREF M2 I J))
              NIEUWE-MATRIX I J))))))))
```

5. Herschrijf de functie MATRIX-SCALAIR-PRODUKT.

```
(DEFUN MATRIX-SCALAIR-PRODUKT (SCALAIR MATRIX)
  ;; zoek eerst de dimensies op van de matrix
  (LET*
    ((DIMS-MATRIX (ARRAY-DIMENSIONS MATRIX))
     (AANTAL-RIJEN (CAR DIMS-MATRIX))
     (AANTAL-KOLOMMEN (CADR DIMS-MATRIX))
     (NIEUWE-MATRIX
       (MAAK-MATRIX AANTAL-RIJEN AANTAL-KOLOMMEN)))
    ;; I en J lopen door de rijen en kolommen
    (DO
      ((I 1 (1+ I)))
      ((= I AANTAL-RIJEN) NIEUWE-MATRIX)
      (DO
        ((J 1 (1+ J)))
        ((= J AANTAL-KOLOMMEN) T)
        ;; het scalair produkt wordt opgeslagen in de nieuwe matrix
        (ASET
          (* SCALAIR (AREF MATRIX I J))
          NIEUWE-MATRIX I J)))))
```

4. STRUCTURES

4.1. DEFINITIES

Een derde belangrijke primitieve datastructuur is de STRUCTURE. Deze komt min of meer overeen met de records van PASCAL-achtige talen. Een structure groepeert een aantal slots. De slots kunnen arbitraire LISP-objecten bevatten. LISP maakt automatisch de nodige constructor, selector- en mutatorfuncties. Bovendien kunnen structures gedeelten van andere structures erven. Hiermee gaan structures in de richting van object-gericht programmeren (zie volgend deel).

Structures worden gemaakt met de functie DEFSTRUCT:

```
(DEFSTRUCT naam-met-opties slots-met-beschrijving)
```

De naam-met-opties is ofwel gelijk aan een naam voor de structure ofwel gelijk aan een lijst waarvan het eerste element de naam is en de overige elementen diverse opties, waarover straks meer. De slots-met-beschrijving is een lijst waarvan elk element een slot beschrijft. De beschrijving van een element is ofwel een atoom dat de naam van het slot aangeeft, ofwel een lijst waarvan het eerste element opnieuw de naam van het slot geeft en het tweede element een initiële waarde.

We zouden bijvoorbeeld een structure voor het object 'persoon' kunnen invoeren:

```
(DEFSTRUCT PERSOON
  LEEFTIJD
  GESLACHT
  BEROEP
  (WOONPLAATS 'BRUSSEL))
```

PERSOON is de naam van de structure. LEEFTIJD, GESLACHT en BEROEP zijn diverse slots. Er werd geen enkele optie gegeven.

Het uitvoeren van een DEFSTRUCT geeft de volgende resultaten:

1. Er wordt automatisch een functie MAKE-naam gemaakt waarbij naam de naam van de structure is. Met deze functies kunnen instanties van de betreffende structure worden gemaakt. De argumenten van deze functie vullen de diverse slots in. De namen van de slots zijn sleutelwoorden in deze functie. Bijvoorbeeld,

```
(SETQ P
      (MAKE-PERSOON :LEEFTIJD 25
                    :GESLACHT 'VROUWELIJK
                    :BEROEP 'COMPUTERGELEERDE))
```

maakt een nieuwe instantie van PERSOON met drie slots ingevuld. Het resultaat wordt gebonden aan de variabele P.

2. Er worden automatisch selectorfuncties gemaakt van de vorm naam-slot. Voor PERSOON is dat PERSOON-LEEFTIJD, PERSOON-GESLACHT, PERSOON-BEROEP. Bijvoorbeeld,

   ```
   (PERSOON-LEEFTIJD P)
   ```

   geeft 25 omdat het LEEFTIJD slot van P gelijk is aan 25.

   ```
   (PERSOON-WOONPLAATS P)
   ```

   geeft BRUSSEL omdat dit als initiële waarde is gegeven bij de definite van PERSOON zelf.

3. De inhoud van een slot kan worden veranderd met de functie SETF. Bijvoorbeeld

   ```
   (SETF (PERSOON-LEEFTIJD P) 26)
   ```

   kent de waarde BRUSSEL toe aan het slot WOONPLAATS van de structure gebonden aan P.

4. Er wordt automatisch een predikaat naam-P gemaakt dat T oplevert als het argument een instantie is van de structuur met de gegeven naam. Bijvoorbeeld,

   ```
   (PERSOON-P P)
   ```

   levert T op.

Bij wijze van voorbeeld bespreken we hier één van de opties voor structures, namelijk :INCLUDE. :INCLUDE is een manier om een vroeger gedefiniëerde structure te gebruiken als onderdeel van de definitie van een andere structure. Bijvoorbeeld

```
(DEFSTRUCT (COMPUTERGELEERDE
              (:INCLUDE PERSOON))
  (BEROEP 'COMPUTER-GELEERDE)
  FAVORIETE-PROGRAMMEERTAAL)
```

definiëert een nieuwe structure met de naam COMPUTERGELEERDE. Deze zal

alle slots van PERSOON erven, en heeft bovendien een nieuw slot, namelijk FAVORIETE-PROGRAMMEERTAAL. Ook wordt het beroep nu geïnitializeerd op COMPUTERGELEERDE.

Andere opties handelen over hoe structures intern zijn geïmplementeerd (bijvoorbeeld met lijsten, met arrays, etc.) of hoe de namen zullen worden gemaakt.

4.2. OEFENINGEN

1. Veronderstel dat we een bevolkingsregister moeten bijhouden. Wat zijn mogelijke datastructuren om dit te doen.

We kunnen een globale variabele *BEVOLKINGSREGISTER* introduceren geïnitializeerd op de lege lijst:

```
(DEFVAR *BEVOLKINGSREGISTER* NIL)
```

De elementen in deze lijst zijn instanties van een structure PERSOON:

```
(DEFSTRUCTURE PERSOON
  NAAM WOONPLAATS LEEFTIJD)
```

Nieuwe personen aan het bestand toevoegen kan nu met de volgende functie:

```
(DEFUN VOEG-PERSOON-TOE
          (NAAM WOONPLAATS LEEFTIJD)
  (SETQ *BEVOLKINGSREGISTER*
    (CONS
      (MAKE-PERSOON NAAM WOONPLAATS LEEFTIJD)
      *BEVOLKINGSREGISTER*)))
```

2. Schrijf een functie voor het veranderen van de leeftijd van een persoon in het register, gegeven de naam van de persoon.

We gebruiken een hulpfunctie die recursief gaat zoeken naar de instantie die de gegeven naam heeft. De wijziging wordt uitgevoerd als die naam gevonden is.

```
(DEFUN VERANDER-LEEFTIJD-2
       (NAAM LEEFTIJD REGISTER)
  (COND
    ((NULL REGISTER) NIL)
    ((EQUAL (PERSOON-NAAM (CAR REGISTER)) NAAM)
      (SETF (PERSOON-LEEFTIJD (CAR REGISTER))
            LEEFTIJD))
    (T
      (VERANDER-LEEFTIJD-2
        NAAM LEEFTIJD (CDR REGISTER)))))
```

VERANDER-LEEFTIJD gebruikt deze hulpfunctie:

```
(DEFUN VERANDER-LEEFTIJD (NAAM LEEFTIJD)
  '(VERANDER-LEEFTIJD-2
    ,NAAM ,LEEFTIJD *BEVOLKINGS-REGISTER*))
```

3. Schrijf een destructieve functie voor het deleteren van een persoon, gegeven de naam.

Eerst een hulpfunctie die de effectieve deletie uitvoert. Merk op dat er twee gevallen zijn. Ten eerste kan het register slechts één persoon bevatten. In dit geval wordt het gehele register terug geïnitializeerd op NIL. Ten tweede is er het geval dat de persoon verderop aanwezig is. Dan gebeurt de destructieve verwijdering van het element.

```
(DEFUN DELETEER-PERSOON-2 (NAAM REGISTER)
  (COND
    ((NULL REGISTER) NIL)
    ((NULL (CDR REGISTER))
      (IF (EQUAL
            (PERSOON-NAAM (SECOND REGISTER)) NAAM)
          (SETQ *BEVOLKINGS-REGISTER* NIL)
          NIL))
    ((EQUAL
       (PERSOON-NAAM (SECOND REGISTER)) NAAM)
      (NCONC LIJST (CDDR REGISTER)))
    (T
      (DELETEER-PERSOON-2 NAAM (CDR REGISTER)))))
```

Deze wordt opgeroepen met DELETEER-PERSOON:

```
(DEFMACRO DELETEER-PERSOON (NAAM)
  '(DELETEER-PERSOON-2
     ,NAAM *BEVOLKINGS-REGISTER*))
```

4. Schrijf een functie om de verzameling oudste inwoners in het bevolkingsregister te vinden.

We schrijven opnieuw een recursieve hulpfunctie die vertrekt met de eerste bewoner als enig lid van de oudsten en eventueel deze hypothese aanpast indien nodig:

```
(DEFUN ZOEK-OUDSTE-INWONER-2 (REGISTER OUDSTEN)
  (COND
    ((NULL REGISTER) OUDSTEN)
    ((> (PERSOON-LEEFTIJD (CAR REGISTER))
        (PERSOON-LEEFTIJD (CAR OUDSTEN)))
      (ZOEK-OUDSTE-INWONER-2
        (CDR REGISTER) (LIST (CAR REGISTER))))
    ((= (PERSOON-LEEFTIJD (CAR REGISTER))
        (PERSOON-LEEFTIJD (CAR OUDSTEN)))
      (ZOEK-OUDSTE-INWONER-2
        (CDR REGISTER)
        (CONS (CAR REGISTER) OUDSTEN)))
    (T
      (ZOEK-OUDSTE-INWONER-2
        (CDR REGISTER) OUDSTEN))))

(DEFMACRO ZOEK-OUDSTE-INWONER ()
  '(ZOEK-OUDSTE-INWONER-2
     (CDR ,*BEVOLKINGS-REGISTER*)
     (LIST (CAR ,*BEVOLKINGS-REGISTER*))))
```

5. Schrijf een functie die het gemiddelde berekent van alle leeftijden in het register. Gebruik een functionele stijl.

```
(DEFUN GEMIDDELDE-LEEFTIJD ()
  (/
    (APPLY '+
           (MAPCAR
             #'(LAMBDA (P) (PERSOON-LEEFTIJD P))
             *BEVOLKINGSREGISTER*))
    (LENGTH *BEVOLKINGSREGISTER*)))
```

5. OBJECTGERICHT PROGRAMMEREN

5.1. DEFINITIES

Generische functies zijn functies die gelden voor meer dan één datatype. Bijvoorbeeld de rekenkundige functie + is een generische functie die zowel voor fixnums als flonums werkt. + zoekt zelf uit wat het type is van de argumenten en kiest de juiste operatie afhankelijk van het type. Generische functies kunnen ook gedefinieerd worden voor abstracte datastructuren.

Veronderstel bijvoorbeeld dat we een teksteditor willen maken met abstracte datastructuren voor een letter, een woord, een zin, een paragraaf, en een tekst. De operaties selecteer, deleteer, verwissel, verplaats, zet om in hoofdletters, enz., zijn generische operaties die in principe uitvoerbaar zijn op elementen van elk type. Als we een letter en een woord van plaats willen verwisselen volstaat het deze dingen te selecteren en te zeggen VERWISSEL. Het systeem zoekt dan zelf uit wat de typen zijn van de argumenten en welke activiteit moet plaats grijpen. Het grote voordeel voor de gebruiker is eenvoud: hij moet enkel maar de generische operaties leren, geen specifieke operaties voor elk type. Het grote voordeel voor de programmeur is dat programma's meer modulair zijn.

Er zijn twee manieren om generische functies te implementeren. Enerzijds kan men een voorwaardelijke uitdrukking gebruiken die nagaat over welk type het gaat en dan de nodige functie oproept. Bijvoorbeeld voor DELETEER krijgen we dan:

```
(DEFUN DELETEER (OBJECT)
  (LET
    ((TYPE (TYPE OBJECT)))
    (COND
      ((EQ TYPE 'LETTER)
        (DELETEER-LETTER OBJECT))
      ((EQ TYPE 'WOORD)
        (DELETEER-WOORD OBJECT))
      ((EQ TYPE 'ZIN)
        (DELETEER-ZIN OBJECT))
      ;; enzovoort voor de andere typen
      ...)))
```

waarbij TYPE een functie is die het type van een object vindt.

Anderzijds kunnen we de functie direct associëren met het type (bijvoorbeeld via de eigenschaplijst) en dan deze functie opzoeken en direct oproepen *zonder* een aparte generische functie te introduceren. Dit geeft aanleiding tot *objectgerichte* programma's die georganiseerd zijn in termen van de objecten (i.e. typen) van het domein en niet in termen van algemene functies zoals DELETEER. Men spreekt soms ook van *datagedreven programma's* omdat de data bepalen welke definitie zal worden uitgevoerd.

Bijvoorbeeld veronderstel:

```
(PUTPROP 'LETTER
         'DELETEER-LETTER
         'DELETEER)

(PUTPROP 'WOORD
         'DELETEER-WOORD
         'DELETEER)
```

enz...

dan is (DELETEER OBJECT) gelijk aan

```
(FUNCALL (GET (TYPE OBJECT) 'DELETEER)
         OBJECT).
```

Dus als het type van OBJECT gelijk is aan LETTER dan is de oproep gelijk aan

```
(FUNCALL 'DELETEER-LETTER OBJECT)
```

Objectgericht programmeren is beter voor het implementeren van generische functies omdat het leidt tot een meer modulair systeem: een nieuw datatype kan worden toegevoegd (bijvoorbeeld hoofdstuk), of de definitie van een datatype kan worden gewijzigd, zonder iets aan de bestaande definities te veranderen. Als we een globale definitie DELETEER gebruiken, moet die telkens herzien worden als er een nieuw type bijkomt.

We kunnen nog een stap verder gaan. In de meeste domeinen zijn er typen die specialisaties zijn van andere typen. Bijvoorbeeld LETTER is een specialisatie van KARAKTER. LEESTEKEN is een andere specialisatie van KARAKTER. Anderzijds zijn HOOFDLETTER en KLEINE-LETTER specialisaties van LETTER. Kleine letters kunnen weer verder onderverdeeld worden in enkelvoudige letters en letters met samengestelde tekens, zoals 'é' of 'ë', die voor sommige functies wellicht een speciale behandeling vereisen. Typen vormen dus een hiërarchie.

Het belang van een hiërarchie is dat een bepaalde definitie met het meest algemene type kan worden verbonden. Bijvoorbeeld de functie DELETEER kan wellicht op het niveau van LETTER of misschien zelfs KARAKTER, worden geplaatst in plaats van een afzonderlijke definitie te introduceren voor HOOFDLETTER, KLEINE-LETTER, enz. We zeggen in dit geval dat de typen HOOFDLETTER en KLEINE-LETTER definities *erven* van het meer algemene type. Het erven van definities is transitief. Als LETTER zelf geen definitie is, zal HOOFDLETTER van KARAKTER erven.

Anderzijds zijn er misschien uitzonderingen waarvoor DELETEER expliciet moet worden gedefinieerd. Bijvoorbeeld samengestelde letters vereisen dat twee of meer tekens worden uitgewist en er zal dus een meer specifieke definitie geassocieerd zijn met samengestelde kleine letter dan met enkelvoudige letter. De definities verbonden met het algemene type zijn dan ook "defaults", ze gelden bij *verstek*. De definitie geldt zolang er geen definitie is lager in de hiërarchie die expliciet de speciale gevallen behandelt voor dit type en zijn specialisaties.

De organisatie van abstracte procedures in een typehiërarchie is een uitzonderlijk krachtig middel om grote systemen met honderden datatypen te structuren. Deze techniek wordt dan ook meer en meer gebruikt, vooral in systeemsoftware, grafische toepassingen en gegevensbestanden.

We kunnen deze ideeën in LISP implementeren door het opzoeken van een definitie ingewikkelder te maken. Veronderstel bijvoorbeeld een functie ZOEK-DEFINITIE die een definitie zoekt in een hiërarchie van typen. ZOEK-DEFINITIE kijkt eerst of

er een definitie is op de eigenschaplijst van het object. Als dit niet het geval is, zoekt ZOEK-DEFINITIE recursief verder bij het type van het object.

```
(DEFUN ZOEK-DEFINITIE (OBJECT FUNCTION)
  (LET
    ((DEFINITIE (GET OBJECT FUNCTION)))
    (COND
     ((AND (NULL DEFINITIE) (NULL (TYPE OBJECT)))
       ;; er is geen definitie en de top van
       ;; de hiërarchie is bereikt
       NIL)
     ((NULL DEFINITIE)
       ;; er is geen definitie, zoek verder
       (ZOEK-DEFINITIE (TYPE OBJECT) FUNCTION))
     (T
       ;; er is een definitie
       DEFINITIE))))
```

Nu definiëren we OFUNCALL, een nieuwe FUNCALL die evaluatie uitvoert voor objectgerichte programma's. OFUNCALL zoekt een definitie met ZOEK-DEFINITIE en voert dan evaluatie uit.

```
(DEFMACRO OFUNCALL (FUNCTIE &REST ARGUMENTEN)
  '(LET
     ((DEFINITIE
       (ZOEK-DEFINITIE (CAR ',ARGUMENTEN) ,FUNCTIE)))
     (COND
       ((NULL DEFINITIE)
        ;; er is geen definitie
        'UNDEFINED)
       (T
        (FUNCALL DEFINITIE ,@ARGUMENTEN)))))
```

Hier is een voorbeeld: veronderstel een (kleine) hiërarchie van getallen met NUMBER als het meest algemene type en FIXNUM en FLONUM als specialisaties. Veronderstel ook

```
(PUTPROP 'FIXNUM '1+ 'ADD1)
(PUTPROP 'FLONUM '1+$ 'ADD1)
```

Dan is

```
(OFUNCALL 'ADD1 10)
```

gelijk aan

```
(FUNCALL '1+ 10)
```

omdat (ZOEK-DEFINITIE 10 'ADD1) gelijk is aan 1+.

Het is gebruikelijk om syntactische vormen te introduceren die het gemakkelijk maken om typen te definiëren. In de komende oefeningen zullen we een functie DEFOBJECT (definieer object) gebruiken met als eerste argument de naam van het type. Het tweede argument geeft het meer algemene type met de syntaxis (EEN meer-algemene-type) en de overige argumenten geven een lijst van associaties tussen functienamen en definities. Bijvoorbeeld de objectgerichte functies van FIXNUM zijn

```
(DEFOBJECT FIXNUM
  (EEN NUMBER)
  (ADD1 '1+)
  (SQRT 'ISQRT)
  ...)
```

In de praktijk zijn de mechanismen voor objectgericht programmeren heel wat ingewikkelder. We hebben het bijvoorbeeld nog niet gehad over situaties waar een type van meer dan één ander type eigenschappen erft of waar er controle op het erven van eigenschappen moet plaats grijpen. Het type speelgoedvrachtwagen erft selectief definities van speelgoed (de prijs, de grootte) èn van vrachtwagen (aantal wielen, uitzicht). De implementatie van objectgerichte programma's en primitieven om dit goed te doen, vormt nog steeds het onderwerp van actief onderzoek. De literatuurlijst op het einde van het boek bevat verdere verwijzingen.

Er zijn talen die een objectgerichte programmeerstijl sterk aanmoedigen. Het bekendste voorbeeld is Smalltalk (Kay, 1972). Zoals we gezien hebben in dit hoofdstuk is het ook mogelijk om objectgericht te programmeren in LISP en moderne LISP-systemen hebben primitieve functies en datastructuren om dit te doen. Twee belangrijke voorbeelden zijn het FLAVOR systeem in Zetalisp en COMMON LOOPS in Interlisp. Deze mechanismen zijn echter nog in volle ontwikkeling zodat geen standaard is ontwikkeld.

5.2. OEFENINGEN

1. Veronderstel dat REEKS en VERZAMELING twee specialisaties zijn van het type LIJST. Definieer generische functies voor AANTAL-ELEMENTEN, VOEG-ELEMENT-TOE, LEEG, EERSTE-ELEMENT, OVERIGE-ELEMENTEN en IS-LID.

```
(DEFOBJECT LIJST
  (AANTAL-ELEMENTEN LENGTH)
  (VOEG-ELEMENT-TOE
    (LAMBDA (LIJST ELEMENT)
      (CONS ELEMENT LIJST)))
  (LEEG NULL)
  (EERSTE-ELEMENT CAR)
  (OVERIGE-ELEMENTEN CDR)
  (IS-LID MEMBER))
```

Bijvoorbeeld, als L van het type LIJST is, dan is (OFUNCALL 'AANTAL-ELEMENTEN L) gelijk aan (FUNCALL 'LENGTH L) en (OFUNCALL 'VOEG-ELEMENT-TOE L) is gelijk aan

```
(FUNCALL
  '(LAMBDA (LIJST ELEMENT)
     (CONS ELEMENT LIJST))
  L)
```

REEKS neemt alle definities van LIJST over:

```
(DEFOBJECT REEKS (EEN LIJST)).
```

VERZAMELING ook, behalve dat VOEG-ELEMENT-TOE verandert:

```
(DEFOBJECT VERZAMELING
  (EEN LIJST)
   (VOEG-ELEMENT-TOE
     (LAMBDA (VERZAMELING ELEMENT)
       (COND
         ((OFUNCALL 'MEMBER VERZAMELING ELEMENT)
           VERZAMELING)
         (T VERZAMELING)))))
```

2. Definieer de functie DEFOBJECT.

DEFOBJECT gebruikt een MAPCAR om de object-specifieke definities te plaatsen op de eigenschaplijst van het object:

```
(DEFMACRO DEFOBJECT (OBJECT &REST ARGS)
  `(PROG (ASSOCIATIES)
    (COND
     ((EQ (CAAR ',ARGS) 'EEN)
      ;; er is een meer algemeen type
      (PUTPROP ',OBJECT (CADAR ',ARGS) 'TYPE)
      (SETQ ASSOCIATIES (CDR ',ARGS))
     (T
      ;; geen algemener type
      (SETQ ASSOCIATIES ',ARGS))))
    (MAPCAR
     #'(LAMBDA (ASSOCIATIE)
        (PUTPROP
          ',OBJECT (CADR ASSOCIATIE) (CAR ASSOCIATIE)))
     ASSOCIATIES)))
```

6. PACKAGES EN MODULES

6.1. DEFINITIES

LISP-programma's worden steeds groter, vooral omdat LISP nu ook als systeemtaal wordt gebruikt. LISP-programma's worden ook meer en meer door teams van programmeurs geschreven. Hierdoor ontstaat de behoefte om grotere eenheden te kunnen hanteren dan de functie. Packages en modules zijn bedoeld om deze behoefte te ondervangen.

6.1.1. PACKAGES

Het package-systeem laat toe om verwarring tussen de namen gebruikt in diverse programma-pakketten te voorkomen en namen (en dus variabelen of functiedefinities) af te schermen voor andere gebruikers. Er zou bijvoorbeeld een programmapakket GRAPHICS kunnen zijn met functies voor het genereren van beelden op het scherm en een ander pakket EDIT met functies voor het editeren van teksten. Beiden kunnen een functie CLEAR-SCREEN hebben. Zonder het package-systeem zou het combineren van beide pakketten een naam-conflict geven. Ook zouden er globale variabelen kunnen zijn die toevallig dezelfde naam hebben

in de twee pakketten. Dit kan tot gevaarlijke interacties leiden als functies uit beide pakketten tegelijk moeten draaien. Met het package-systeem kan men deze problemen vermijden.

Een package heeft twee soorten namen:

1. *Interne namen* die volledig binnen een package liggen en in principe niet toegankelijk zijn van buitenaf. De symbolen aangeduid door deze namen noemt men *interne symbolen.*

2. *Externe namen* die ook vanuit een ander package kunnen worden gebruikt om symbolen binnen het package te adresseren. Men noemt deze symbolen *externe symbolen.* Externe namen zijn een soort interface tussen het package en de rest van het systeem.

Elk package heeft een naam. EDIT of GRAPHICS zijn voorbeelden. We bekijken eerst hoe packages worden gebruikt en dan hoe ze worden gemaakt.

GEBRUIK VAN PACKAGES

Normaal zijn er op zijn minst de volgende packages:

1. LISP: Dit package bevat externe symbolen voor de primitieve functies van het LISP systeem, zoals CAR en CONS, en de globale variabelen. De functies en variabelen die intern zijn aan de LISP-implementatie zelf zijn hierdoor beschermd.

2. USER: Dit package bevat de functies die door de gebruiker zijn ingevoerd, zonder dat hij extra packages gebruikt.

3. SYS (afkorting van 'system'): Dit package bevat de functies die nodig zijn om de volledige LISP omgeving te doen functioneren.

4. KEYWORD: Dit package bevat symbolen waarvoor er geen naamconflicten kunnen zijn. Dit is vooral het geval bij de sleutelwoorden in functies. Zij dienen enkel als documentatie en niet als namen van objecten binnen een package.

Packages kunnen symbolen van elkaar erven. Bijvoorbeeld binnen het USER package zal het af en toe nodig zijn om LISP-functies te gebruiken die zich binnen het LISP package bevinden. Door een ervingsrelatie te definiëren vanuit het USER package naar het LISP package worden automatisch alle externe symbolen van het LISP package externe symbolen van het USER package.

Symbolen binnen een package worden als volgt geadresseerd.

1. Tijdens de evaluatie is er altijd een current-package (gebonden aan de globale variabele *PACKAGE*). Wanneer een symbool moet worden geadresseerd binnen dit package of binnen één van de packages waar het van erft moet er niets speciaals gebeuren. Bijvoorbeeld, FOO is de naam van een symbool binnen het current package. Wanneer een LISP-systeem opstart is USER gewoonlijk het current package.
2. Symbolen binnen het keyword package zijn addresseerbaar vanuit gelijk welk package door een ':' te plaatsen voor het symbool. Bovendien worden al de symbolen in het keyword package als konstanten beschouwd, die naar zichzelf evalueren. Dus :FOO evalueert altijd naar het symbool :FOO.
3. Het is verder mogelijk om een extern symbool in een ander package te addresseren via de vorm package-naam:symbool-naam. Bijvoorbeeld, GRAPHICS:CLEAR-SCREEN addresseert het symbool CLEAR-SCREEN in het GRAPHICS-package, terwijl EDITOR:CLEAR-SCREEN het symbool CLEAR-SCREEN in het editor-package adresseert.
4. Tenslotte wordt een intern symbool gerefereerd door de vorm package-naam::symbool-naam. Bijvoorbeeld, GRAPHICS::*MY-SCREEN* adresseert het interne symbool *MY-SCREEN* in het GRAPHICS-package. Dit symbool is niet adresseerbaar als men zich niet in het GRAPHICS package bevindt.

DEFINITIE VAN PACKAGES

Packages zijn in feite abstracte datastructuren die de band leggen tussen een symbool en zijn interne referent. Hier zijn twee belangrijke functies om packages te definiëren, gevolgd door enkele andere functies die met packages werken:

1. (MAKE-PACKAGE package-name)
   maakt een package met de gegeven package-name.
2. (IN-PACKAGE package-name)
   zet het current package gelijk aan het package met package-name. Als dat nog niet bestaat, wordt eerst de functie MAKE-PACKAGE opgeroepen om het te creëren.
3. (USE-PACKAGE packages)
   voegt de gegeven packages toe aan de lijst van packages waaruit de current package erft.

4. (EXPORT list-of-symbols &OPTIONAL package)
   maakt de symbolen in de list-of-symbols externe symbolen van het gegeven package. Als er geen package gegeven is wordt het current package verondersteld.

5. (IMPORT list-of-symbols &OPTIONAL package)
   importeert de symbolen in de list-of-symbols als interne symbolen van het gegeven package. De default is opnieuw current package.

EXPORT en IMPORT zijn een meer gecontroleerde manier om symbolen van het ene naar het andere package te transfereren dan USE-PACKAGE.

Bijvoorbeeld, de volgende reeks formules:

```
(MAKE-PACKAGE 'GRAPHICAL-EDITOR)
(USE-PACKAGE '(GRAPHICS EDITOR))
```

maakt een nieuw package GRAPHICAL-EDITOR die alle externe symbolen erft van de packages GRAPHICS en EDITOR.

```
(EXPORT '(GRAPHICAL-EDIT *GRAPHICAL-EDIT-WINDOW* ...))
```

zorgt ervoor dat GRAPHICAL-EDIT, *GRAPHICAL-EDIT-WINDOW*, enz. adresseerbaar zijn als externe symbolen in de current package.

```
(IMPORT '(GRAPHICS:CLEAR-SCREEN EDITOR:DELETE-LINE))
```

zorgt ervoor dat de symbolen CLEAR-SCREEN en DELETE-LINE uit de packages GRAPHICS en EDITOR nu als symbolen binnen de current package bekend zijn.

6.1.2. MODULES

Een module is een deelsysteem dat normaal bestaat uit een reeks van files waarop allerlei functies zijn gedefiniëerd en variabelen gedeclareerd. Modules komen dikwijls overeen met packages. Een module heeft een naam. De twee belangrijkste functies voor het hanteren van modules zijn:

1. (PROVIDE module-naam) maakt de module met de gegeven naam 'aanwezig' in de LISP-omgeving.
2. (REQUIRE module-naam) zegt dat de module met de gegeven naam aanwezig moet zijn in de LISP-omgeving.

Hier is een voorbeeld van een initialisatie file voor een module die de semantische component van een taalsysteem bevat.

```
;;; zowel de module als het package heten SEMANTICS

(PROVIDE 'SEMANTICS)
(IN-PACKAGE 'SEMANTICS)

;;; de modules voor syntaxis en morfologie moeten geladen zijn:

(REQUIRE 'SYNTAX)
(REQUIRE 'MORPHOLOGY)

;;; er is ook een module TREE om bomen te manipuleren die
;;; moet geladen zijn:

(REQUIRE 'TREE)

;;; alle symbolen uit het TREE package komen automatisch in
;;; het SEMANTICS package:

(USE-PACKAGE 'TREE)

;;; importeer de symbolen *SYNTAX-TREE* uit het package SYNTAX
;;; en *DICTIONARY* uit het package MORPHOLOGY:

(IMPORT
  '(SYNTAX:*SYNTAX-TREE* MORPHOLOGY:*DICTIONARY*))

;;; hier de diverse oproepen voor het laden van de files
;;; die de functies en andere onderdelen van de SEMANTICS
;;; module beschrijven

...
```

7. SAMENVATTING

In dit deel hebben we enkele belangrijke datastructuren bestudeerd in LISP (de eigenschaplijst en de array) en gezien hoe abstracte datastructuren kunnen gedefinieerd worden in LISP.

GET, PUTPROP en REMPROP zijn functies voor eigenschaplijsten verbonden met atomen.

MAKE-ARRAY, AREF, ASET en ARRAY-DIMENSIONS zijn de belangijkste functies voor arrays.

Verder hebben we gezien hoe we een objectgerichte programmeerstijl in LISP konden implementeren.

Modules en Packages zijn de belangrijkste hulpmiddelen die beschikbaar zijn om grote programma's op een modulaire manier te ontwikkelen.

8. GEMENGDE OPGAVEN

Zoals in de vorige delen bekijken we nu een aantal gemengde opgaven die de functies en concepten uit dit deel illustreren aan de hand van meer complexe voorbeelden. Eerst de opgaven en dan de oplossingen. Een laatste paragraaf bevat nog enkele bijkomende opgaven die niet verder worden opgelost.

8.1. DE OPGAVEN

8.1.1. EEN DATASTRUCTUUR VOOR RATIONELE GETALLEN

Een rationeel getal $\frac{n}{d}$ heeft twee componenten: een numerator n en een denominator d. Ontwerp een abstracte datastructuur voor rationele getallen. Schrijf een functie (RAT x y) die een rationeel getal $\frac{x}{y}$ samenstelt, een functie NUMERATOR die de numerator geeft van een rationeel getal, een functie DENOMINATOR die de denominator geeft, en enkele predikaten en rekenkundige functies: +RAT, -RAT, *RAT, /RAT, =RAT.

Veronderstel eerst dat rationele getallen voorgesteld worden als lijsten. Verander dan de representatie naar een array met twee elementen. Gebruik zoveel mogelijk macro's.

8.1.2. GENEREREN MET HERSCHRIJFGRAMMATICA'S

Het is mogelijk om een taal (bvb. $\{a^n b^n \mid n \geq 1\}$) te beschrijven met een stel herschrijfregels zoals:

```
S → a S b
S → a b
```

Symbolen, zoals S, die links van de pijl voorkomen heten *nonterminale* symbolen. Symbolen die alleen rechts van de pijl voorkomen, zoals a en b, heten *terminale* symbolen.

Een uitdrukking kan *herschreven* worden door een symbool dat links van de pijl voorkomt te vervangen door de reeks letters rechts van de pijl. Als begonnen

wordt met S en als er geen nonterminale symbolen meer overblijven dan is de uitdrukking in de taal beschreven door de grammatica. Bijvoorbeeld 'aabb' is in de taal beschreven door de bovenstaande grammatica omdat we de volgende herschrijfstappen kunnen hebben:

S ⇒ a S b ⇒ a a b b

Gegeven een stel herschrijfregels, schrijf een programma dat alle zinnen van een bepaalde lengte genereert. Ontwerp eerst de nodige abstracte datastructuren. Gebruik zoveel mogelijk macro's.

8.1.3. EEN TYPE-HIERARCHIE VOOR GEOMETRISCHE OBJECTEN

In de meetkunde kunnen de veelhoeken in een hiërarchie worden georganiseerd op basis van het aantal zijden. Construeer een hiërarchie die de objecten driehoek, trapezium, parallellogram, rechthoek en vierkant bevat. Definieer de volgende generische functies:

1. AANTAL-ZIJDEN berekent het aantal zijden.
2. AANTAL-HOEKEN berekent het aantal hoeken.
3. OPPERVLAKTE berekent de oppervlakte.
4. OMTREK berekent de omtrek.

Veronderstel functies die de zijden, de lengte van de zijden, de basis, hoogte, etc. van een veelhoek berekenen. (Deze informatie kan bijvoorbeeld op de eigenschaplijsten van een veelhoek worden geplaatst.)

Bijvoorbeeld, (OMTREK V-1) waarbij V-1 van het type vierkant is met de lengte van een zijde gelijk aan 2, is gelijk aan 8.

8.2. OPLOSSINGEN

8.2.1. EEN DATASTRUCTUUR VOOR RATIONELE GETALLEN

Eerst de primitieve constructor- en selectorfuncties voor een lijstrepresentatie:

```
(DEFMACRO RAT (NUMERATOR DENOMINATOR)
  '(LIST ,NUMERATOR ,DENOMINATOR))

(DEFMACRO NUMERATOR (X)
  '(CAR ,X))
```

```
(DEFMACRO DENOMINATOR (X)
  '(CADR ,X))
```

Nu de predikaten en rekenkundige functies:

De som van twee rationele getallen $\frac{n}{d}$ en $\frac{n'}{d'}$ is gelijk aan

$$\frac{n}{d} + \frac{n'}{d'} = \frac{nd' + n'd}{d\,d'}$$

en dus

```
(DEFMACRO +RAT (X Y)
  '(LET
    ((X ,X)
     (Y ,Y)
    (RAT
      (+ ( * (NUMERATOR X) (DENOMINATOR Y))
         ( * (DENOMINATOR X) (NUMERATOR Y)))
      ( * (DENOMINATOR X) (DENOMINATOR Y))))))
```

Merk op hoe de +RAT macro andere macro's gebruikt.

Het verschil van twee rationele getallen $\frac{n}{d}$ en $\frac{n'}{d'}$ is gelijk aan

$$\frac{n}{d} - \frac{n'}{d'} = \frac{nd' - n'd}{d\,d'}$$

en dus

```
(DEFMACRO -RAT (X Y)
  '(LET
    ((X ,X)
     (Y ,Y))
    (RAT
      (- ( * (NUMERATOR X) (DENOMINATOR Y))
        ( * (DENOMINATOR X) (NUMERATOR Y)))
      ( * (DENOMINATOR X) (DENOMINATOR Y)))))
```

Het produkt van twee rationele getallen $\frac{n}{d}$ en $\frac{n'}{d'}$ is gelijk aan

$$\frac{n}{d} \times \frac{n'}{d'} = \frac{n\,n'}{d\,d'}$$

en dus:

```
(DEFMACRO *RAT (X Y)
  '(LET
    ((X ,X)
     (Y ,Y))
   (RAT
    (* (NUMERATOR X) (NUMERATOR Y))
    (* (DENOMINATOR X) (DENOMINATOR Y)))))
```

Het quotiënt van twee rationele getallen $\frac{n}{d}$ en $\frac{n'}{d'}$ is gelijk aan

$$\frac{n}{d} / \frac{n'}{d'} = \frac{n\,d'}{d\,n'}$$

en dus

```
(DEFMACRO /RAT (X Y)
  '(LET
    ((X ,X)
     (Y ,Y))
    (RAT
     (* (NUMERATOR X) (DENOMINATOR Y))
     (* (DENOMINATOR X) (NUMERATOR Y)))))
```

Twee rationele getallen $\frac{n}{d}$ en $\frac{n'}{d'}$ zijn gelijk als

$$n \times d' = n' \times d$$

```
(DEFMACRO =RAT (X Y)
 '(LET
   ((X ,X)
    (Y ,Y))
   (= (* (NUMERATOR X) (DENOMINATOR Y))
      (* (DENOMINATOR X) (NUMERATOR Y)))))
```

Nu veranderen we de onderliggende representatie van een lijst naar een array:

```
(DEFMACRO RAT (NUMERATOR DENOMINATOR)
 '(LET
   ((NEW-RAT
     (MAKE-ARRAY 1 'FIXNUM 2)))
   (ASET ,NUMERATOR NEW-RAT 1 1)
   (ASET ,DENOMINATOR NEW-RAT 1 2)
   NEWRAT))
```

```
(DEFMACRO NUMERATOR (X)
   '(AREF ,X 1 1))

(DEFMACRO DENOMINATOR (X)
   '(AREF ,X 1 2))
```

De rekenkundige functies en de predikaten blijven ongewijzigd.

8.2.2. GENEREREN MET HERSCHRIJFGRAMMATICA'S

We beginnen met een datastructuur te ontwerpen om grammatica's intern te representeren. We veronderstellen dat er slechts één grammatica is, en dat de regels bijgehouden worden op een globale variabele *REGELS*. De nonterminale symbolen worden bijgehouden op een globale variabele *NONTERMINALEN*. Om een grammatica te definiëren introduceren we een functie DEFGRAMMAR die deze globale variabelen waarden geeft. Bijvoorbeeld na

```
(DEFGRAMMAR
   (S → a S b)
   (S → a b)
   (S → b S a))
```

is *REGELS* gelijk aan

```
((S → a S b)
 (S → a b)
 (S → b S a))
```

en *NONTERMINALEN* gelijk aan (S).

DEFGRAMMAR wordt als een macro gedefinieerd, omdat de argumenten niet moeten geëvalueerd worden:

```
(DEFMACRO DEFGRAMMAR (&REST ARGS)
  '(PROGN
     (SETQ *REGELS* ',ARGS)
     (SETQ *NONTERMINALEN*
           (NONTERMINALS *REGELS* NIL))))
```

NONTERMINALS is een hulpfunctie die de nonterminale symbolen uit een stel regels haalt:

```
(DEFUN NONTERMINALS (REGELS RESULTAAT)
  (COND
    ((NULL REGELS) RESULTAAT)
    ((MEMBER (CAAR REGELS) RESULTAAT)
      (NONTERMINALS (CDR REGELS) RESULTAAT))
    (T (NONTERMINALS (CDR REGELS)
                          (CONS (CAAR REGELS) RESULTAAT)))))
```

Hier zijn nog enkele hulpfuncties die mede de abstracte datastuctuur voor regels uitmaken. De functie LINKERZIJDE levert de linkerzijde van een regel en de functie RECHTERZIJDE de rechterzijde:

```
(DEFMACRO LINKERZIJDE (REGEL)
  '(CAR ,REGEL))

(DEFMACRO RECHTERZIJDE (REGEL)
  '(CDDR ,REGEL))
```

De functie NONTERMINAAL bepaalt of een symbool nonterminaal is of niet

```
(DEFMACRO NONTERMINAAL (SYMBOL)
  '(MEMBER ,SYMBOL *NONTERMINALEN*))
```

Nu een functie HERSCHRIJF die het eerste nonterminaal symbool in een uitdrukking herschrijft volgens een regel van de grammatica. Als de regel niet gebruikt kan worden voor de uitdrukking is het resultaat NIL. Bijvoorbeeld (HERSCHRIJF '(S) '(S → a S b) NIL) is gelijk aan (a S b).

Er zijn drie gevallen:

1. De uitdrukking is NIL, in dit geval is het resultaat NIL.
2. De uitdrukking begint met een nonterminaal symbool. Als dit nonterminaal symbool gelijk is aan de linkerzijde van de regel, dan is het resultaat gelijk aan de uitdrukking met het nonterminaal symbool vervangen door de rechterzijde. Anders is het resultaat gelijk aan NIL.
3. De uitdrukking begint niet met een nonterminaal symbool. In dit geval moeten we recursief de functie toepassen.

De elementen voor het nonterminale symbool worden bijgehouden op een variabele OUDE-UITDRUKKING:

```
(DEFUN HERSCHRIJF
        (UITDRUKKING REGEL OUDE-UITDRUKKING)
  (COND
    ((NULL UITDRUKKING) NIL)
    ((NONTERMINAAL (CAR UITDRUKKING))
      (COND
        ((EQ (CAR UITDRUKKING) (LINKERZIJDE REGEL))
          (APPEND OUDE-UITDRUKKING
                  (RECHTERZIJDE REGEL) (CDR UITDRUKKING)))
        (T NIL)))
    (T (HERSCHRIJF (CDR UITDRUKKING)
                   REGEL
                   (APPEND
                     OUDE-UITDRUKKING
                     (LIST (CAR UITDRUKKING)))))))
```

De volgende functie, HERSCHRIJF*, past HERSCHRIJF toe op een reeks van regels. Het resultaat is een lijst van mogelijke herschrijvingen van een uitdrukking. Bijvoorbeeld (HERSCHRIJF* '(S) *REGELS*) is gelijk aan ((a S b) (a b) (b S a)).

```
(DEFUN HERSCHRIJF* (UITDRUKKING REGELS)
  (COND
    ((NULL REGELS) NIL)
    (T
     (LET
       ((NIEUWE-UITDRUKKING
          (HERSCHRIJF UITDRUKKING (CAR REGELS) NIL)))
       (COND
         ((NULL NIEUWE-UITDRUKKING)
           (HERSCHRIJF* UITDRUKKING (CDR REGELS)))
         (T (CONS
              NIEUWE-UITDRUKKING
              (HERSCHRIJF* UITDRUKKING (CDR REGELS)))))))))
```

Tenslotte nog een functie *HERSCHRIJF* die HERSCHRIJF* toepast op een reeks van uitdrukkingen. Bijvoorbeeld (*HERSCHRIJF* '((S))) is gelijk aan ((a S b) (a b) (b S a)).

```
(DEFUN *HERSCHRIJF* (REEKS-UITDRUKKINGEN)
  (COND
    ((NULL REEKS-UITDRUKKINGEN) NIL)
    (T
      (APPEND
        (HERSCHRIJF* (CAR REEKS-UITDRUKKINGEN) *REGELS*)
        (*HERSCHRIJF* (CDR REEKS-UITDRUKKINGEN))))))
```

Deze functie kan nu dienen om een generator te maken. De generator begint met S te herschrijven. Als er uitdrukkingen zijn in de verzameling waarvan de lengte gelijk is aan de gevraagde lengte hebben we het antwoord gevonden. Als de uitdrukkingen korter zijn genereren we een nieuwe reeks uitdrukkingen. Als de uitdrukkingen langer zijn dan is het resultaat NIL omdat er geen kortere uitdrukkingen meer zullen komen:

```
(DEFUN GENERATOR (LENGTE BEGIN-UITDRUKKINGEN)
  (LET
    ((REEKS (*HERSCHRIJF* BEGIN-UITDRUKKINGEN)))
    (COND
      ((NULL REEKS) NIL)
      (T
        (LET
          ((GELDIGE-UITDRUKKINGEN
            (FILTER-GELDIGE-UITDRUKKINGEN REEKS LENGTE)))
          (COND
            (GELDIGE-UITDRUKKINGEN GELDIGE-UITDRUKKINGEN)
            ((VERDER-GAAN REEKS LENGTE)
              (GENERATOR LENGTE REEKS))
            (T NIL)))))))
```

FILTER-GELDIGE-UITDRUKKINGEN is een functie die de terminale uitdrukkingen van een bepaalde lengte haalt uit een verzameling uitdrukkingen:

```
(DEFUN FILTER-GELDIGE-UITDRUKKINGEN (REEKS LENGTE)
  (COND
    ((NULL REEKS) NIL)
    ((AND
       (EQ (LENGTH (CAR REEKS)) LENGTE)
       (ALLEEN-TERMINALEN (CAR REEKS)))
     (CONS
       (CAR REEKS)
       (FILTER-GELDIGE-UITDRUKKINGEN
         (CDR REEKS) LENGTE)))
    (T
      (FILTER-GELDIGE-UITDRUKKINGEN
        (CDR REEKS) LENGTE))))
```

waarbij ALLEEN-TERMINALEN nagaat of een uitdrukking alleen uit terminalen bestaat of niet:

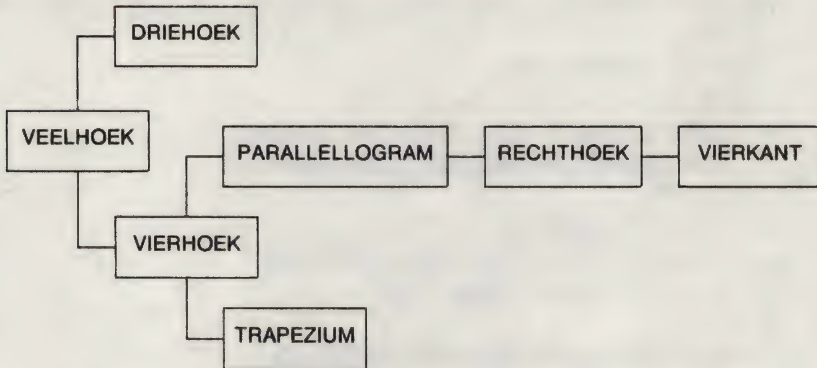
```
(DEFUN ALLEEN-TERMINALEN (UITDRUKKING)
  (COND
    ((NULL UITDRUKKING) T)
    ((NONTERMINAAL (CAR UITDRUKKING)) NIL)
    (T (ALLEEN-TERMINALEN (CDR UITDRUKKING)))))
```

Het predikaat VERDER-GAAN bepaalt of we nog verder moeten gaan met een bepaalde reeks voor een gegeven lengte. Immers als de terminale uitdrukkingen langer zijn dan de gevraagde lengte weten we dat er geen juiste oplossingen meer zullen komen:

```
(DEFUN VERDER-GAAN (REEKS LENGTE)
  (COND
    ((NULL REEKS) T)
    ((AND (> (LENGTH (CAR REEKS)) LENGTE)
          (ALLEEN-TERMINALEN (CAR REEKS)))
     NIL)
    (T
      (VERDER-GAAN (CDR REEKS) LENGTE))))
```

8.2.3. EEN TYPE-HIERARCHIE VOOR GEOMETRISCHE OBJECTEN

De hiërarchie van veelhoeken ziet er als volgt uit:



Het is de bedoeling om de functies op de juiste plaats aan te brengen in de hiërarchie. AANTAL-ZIJDEN en AANTAL-HOEKEN kunnen komen op het niveau van DRIEHOEK en VIERHOEK. OMTREK kan op het niveau van VEELHOEK. OPPERVLAKTE komt op het niveau van DRIEHOEK, PARALLELLOGRAM en TRAPEZIUM.

Hier zijn dan de definities zelf. OMTREK wordt gedefinieerd op het niveau van VEELHOEK. De omtrek is gelijk aan de som van de lengte van de zijden:

```
(DEFOBJECT VEELHOEK
   (OMTREK
     (LAMBDA (V)
       (APPLY '+
         (MAPCAR
           '(LAMBDA (ZIJDE)
              (LENGTE ZIJDE))
         (ZIJDEN V))))))
```

DRIEHOEK erft OMTREK van VEELHOEK maar heeft zelf AANTAL-ZIJDEN, AANTAL-HOEKEN en OPPERVLAKTE:

```
(DEFOBJECT DRIEHOEK
  (EEN VEELHOEK)
  (AANTAL-ZIJDEN (CONSTANTE 3))
  (AANTAL-HOEKEN (CONSTANTE 3))
  (OPPERVLAKTE
    (LAMBDA (V)
      (/
        (* (BASIS V) (HOOGTE V))
        2))))
```

CONSTANTE is een functie die een constante geeft voor gelijk welk argument (cf. oefening 11 pag. 4.22).

VIERHOEK erft definities van VEELHOEK maar heeft zelf AANTAL-ZIJDEN en AANTAL-HOEKEN:

```
(DEFOBJECT VIERHOEK
  (EEN VEELHOEK)
  (AANTAL-ZIJDEN (CONSTANTE 4))
  (AANTAL-HOEKEN (CONSTANTE 4)))
```

PARALLELLOGRAM bevat de definitie van OPPERVLAKTE en erft de rest van VIERHOEK:

```
(DEFOBJECT PARALLELLOGRAM
  (EEN VIERHOEK)
  (OPPERVLAKTE
    (LAMBDA (V)
      (* (BASIS V) (HOOGTE V)))))
```

RECHTHOEK en VIERKANT erven al hun definities:

```
(DEFOBJECT RECHTHOEK (EEN PARALLELLOGRAM))

(DEFOBJECT VIERKANT (EEN RECHTHOEK))
```

TRAPEZIUM berekent de oppervlakte anders dan VIERHOEK, maar erft de rest van de definities.

```
(DEFOBJECT TRAPEZIUM
  (EEN VIERHOEK)
  (OPPERVLAKTE
    (LAMBDA (V)
      (/
        (+ (GROTE-BASIS V) (KLEINE-BASIS V))
        (* 2 (HOOGTE V))))))
```

Bijvoorbeeld veronderstel een rechthoek R-1 met de volgende eigenschappen:

```
(PUTPROP 'R-1 'RECHTHOEK 'TYPE)
(PUTPROP 'R-1 15 'BASIS)
(PUTPROP 'R-1 10 'HOOGTE)
```

Dan is (OFUNCALL OPPERVLAKTE 'R-1) gelijk aan

```
(FUNCALL
   '(LAMBDA (V)
      (* (BASIS V) (HOOGTE V)))
   'R-1)
```

Dit geeft na lambda-conversie

```
(* (BASIS 'R-1) (HOOGTE 'R-1))
```

en dus 150.

8.3. NOG ENKELE OPGAVEN

Schrijf een functie voor het produkt van twee matrices, waarbij de matrices als arrays zijn voorgesteld.

Schrijf een functie die nagaat of een uitdrukking deel uitmaakt van de taal beschreven door een herschrijfgrammatica.

Ontwerp een datastructuur voor veeltermen. Schrijf enkele functies zoals som, verschil, deling en produkt van veeltermen.

Werk het meetkundig voorbeeld verder uit. Ontwerp datastructuren voor de andere eigenschappen van een veelhoek en de componenten van een veelhoek.

DEEL 6. EVALUATIE

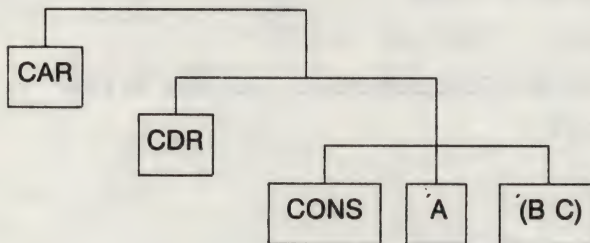
1. HET EVALUATIE PRINCIPE.
2. EEN EVALUATOR VOOR LISP IN LISP.
3. EEN EVALUATOR VOOR EEN REGELSYSTEEM IN LISP.
4. SAMENVATTING

Evaluatie is de kern van LISP. In dit deel bekijken we evaluatie even van dichterbij. De beste manier om dit te doen is door een evaluator in LISP zelf te definiëren. Dan bekijken we hoe we zelf nieuwe talen in LISP kunnen implementeren. We zullen een regelsysteem, dat ondermeer bij expertsystemen veel voorkomt, in detail bestuderen.

1. EVALUATIE

1.1. DEFINITIES

Intuïtief kunnen we een formule beschouwen als een boom. Bijvoorbeeld (CAR (CDR (CONS 'A '(B C)))) is gelijk aan de boom



De meest linkse tak in elke knoop bevat de functie die moet toegepast worden op evaluatie van de parallelle rechtertakken. Het evaluatieproces doorkruist de boom te beginnen met de top en daalt af naar de laagste knopen. De evaluator klimt dan terug naar de top, door de functie in de meest linkse knoop toe te passen. De functie EVAL realiseert dit proces.

EVAL heeft twee argumenten: een LISP-object en een a-lijst (associatie-lijst) die de bindingen bevat die op dit moment gelden. De a-lijst is een lijst van lijsten met elke deellijst een paar. Het eerste element is de naam van de variabele, het tweede element de binding. Bijvoorbeeld ((L (A B)) (M NIL)) is een a-lijst. L is geassocieerd met (A B) en M met NIL.

De evaluatie zelf bevat de volgende stappen.

1. Als het LISP-object gelijk is aan de nulllijst is het resultaat van evaluatie de nulllijst. Bijvoorbeeld (EVAL NIL A-LIJST) is gelijk aan NIL.
2. Als het LISP-object gelijk is aan een getal of het speciale atoom T is het resultaat het LISP-object zelf. Bijvoorbeeld (EVAL 12 A-LIJST) is gelijk aan 12.

3. Als het LISP-object gelijk is aan een atoom is het resultaat gelijk aan de binding van dit atoom in de a-lijst. Bijvoorbeeld (EVAL A '((A (B C)))) is gelijk aan (B C) omdat de binding van A op de a-lijst gelijk is aan (B C).
4. Als het LISP-object een lijst is en het eerste element van de lijst is het atoom QUOTE, dan is evaluatie gelijk aan het tweede element van de lijst. Bijvoorbeeld (EVAL S A-LIJST) met S gelijk aan 'A of (QUOTE A) is gelijk aan A.
5. Als het LISP-object gelijk is aan een lijst en het eerste element is een primitieve functie (zoals CAR of CDR) dan wordt de functie toegepast op het resultaat van de evaluatie van de argumenten. Bijvoorbeeld (EVAL S A-LIJST) met S gelijk aan (CAR '(A B C)) is gelijk aan A omdat de evaluatie van '(A B C) gelijk is aan (A B C) (geval 4) en CAR toepast op (A B C) gelijk is aan A.
6. Tenslotte als het LISP-object gelijk is aan een lijst en het eerste element is een gedefinieerde functie, dan moet EVAL recursief toegepast worden. Het LISP-object in de nieuwe EVAL is gelijk aan de functiedefinitie en de a-lijst bevat de argumenten in de functiedefinitie verbonden met het resultaat van de evaluatie van de argumenten in de eerste LISP-object. Bijvoorbeeld (EVAL S A-LIJST) met S gelijk aan (SECOND '(A B C)) leidt tot (EVAL S A-LIJST) met S gelijk aan (CAR (CDR L)) en de a-lijst gelijk aan ((L (A B C))).

1.2. OEFENINGEN

1. Gegeven L gelijk aan (A B) hoe begint de evaluatie voor (CAR L)?

De evaluatie begint met (EVAL S A-LIJST), waarbij S gelijk is aan (CAR L) en A-LIJST gelijk aan ((L (A B))).

2. Wat is de eerste stap?

Het eerste element van het LISP-object is een primitieve functie: CAR. We moeten CAR toepassen op het resultaat van de evaluatie van het argument L, we krijgen dus een recursieve oproep van eval: (EVAL S A-LIJST) met S gelijk aan L en A-LIJST gelijk aan ((L (A B))).

3. Wat is het resultaat van de evaluatie van L?

(A B). Als het LISP-object een atoom is (maar geen getal) is het resultaat van evaluatie gelijk aan de binding van dit atoom op de a-lijst. De a-lijst is gelijk aan ((L (A B))) en dus het resultaat is gelijk aan (A B).

4. Wat gebeurt er nu?

We moeten CAR toepassen op dit resultaat. Het eerste element van (A B) is gelijk aan A, dus (CAR L) is gelijk aan A.

5. We gaan nu bekijken hoe de evaluatie werkt voor

```
(DEFINE SYMBOL-TEST (L)
  (COND
    ((NULL L) T)
    ((SYMBOL (CAR L)) (SYMBOL-TEST (CDR L)))
    (T NIL)))
```

toegepast op (SYMBOL-TEST M) met M gelijk aan (B C). Hoe begint de evaluatie?

De evaluatie begint met (EVAL S A-LIJST) waarbij A-LIJST gelijk is aan ((M (B C))) en S gelijk aan (SYMBOL-TEST M).

6. Wat is de eerste stap in de evaluatie?

Het LISP-object is een lijst en het eerste element is een gedefinieerde functie. Dus moeten we een nieuwe EVAL uitvoeren waarbij het LISP-object gelijk is aan de functiedefinitie en de a-lijst bevat de bindingen tussen de argumenten in de functiedefinitie (hier L) en de evaluatie van de argumenten in het LISP-object.

7. Wat is de nieuwe a-lijst?

((L (B C))). We moeten de binding tussen L en de evaluatie van M toevoegen aan de a-lijst. M is een atoom, dus de evaluatie van M is gelijk aan de waarde verbonden met M in de associatielijst, d.w.z. (B C). De binding tussen L en (B C) geeft als nieuwe a-lijst ((L (B C))).

8. Wat is het beginpunt in de nieuwe evaluatie van EVAL?

(EVAL S A-LIJST), met S gelijk aan

```
(COND
  ((NULL L) T)
  ((SYMBOL (CAR L)) (SYMBOL-TEST (CDR L)))
  (T NIL))
```

en A-LIJST gelijk aan ((L (B C))). S komt uit de functiedefinitie van SYMBOL-TEST.

9. Welke functie in de voorwaardelijke uitdrukking wordt nu uitgevoerd?

___

(SYMBOL-TEST (CDR L)), want L is geen nulllijst en (CAR L) is een atoom.

10. Wat is het beginpunt voor de volgende stap in de evaluatie?

___

(EVAL S A-LIJST) met S gelijk aan (SYMBOL-TEST (CDR L)) en A-LIJST gelijk aan ((L (B C))).

11. Wat is de volgende oproep voor EVAL?

___

(EVAL S A-LIJST) met S gelijk aan

```
(COND
  ((NULL L) T)
  ((SYMBOL (CAR L)) (SYMBOL-TEST (CDR L)))
  (T NIL))
```

en A-LIJST gelijk aan ((L (C))). S komt opnieuw uit de functiedefinitie van SYMBOL-TEST en de nieuwe binding (L (C)) komt door de argumenten in de functiedefinitie, namelijk L, te verbinden met het resultaat van het argument, namelijk (CDR L). L was gelijk aan (B C) en (CDR L) is dus gelijk aan (C).

12. Wat is het resultaat van deze stap in de evaluatie?

___

Een nieuwe evaluatie van SYMBOL-TEST met L gelijk aan NIL.

13. Wat is de a-lijst voor deze evaluatie?

___

((L NIL)) omdat de (CDR L) voor ((L (C))) gelijk is aan NIL.

14. Wat is het resultaat van de evaluatie?

T want (NULL L) is T in ((L NIL)).

15. Wat is het eindresultaat van (SYMBOL-TEST M) met M gelijk aan (B C)?

T, er zijn immers geen verdere uitdrukkingen die geëvalueerd moeten worden.

2. DE EVALUATOR

Hier is de definitie van EVAL:

```
(DEFINE EVAL (S A-LIJST)
  (COND
    ((NULL S) NIL)
    ((NUMBERP S) S)
    ((EQ S T) S)
    ((SYMBOLP S) (ASSOC S A-LIJST))
    ((EQ (CAR S) 'QUOTE) (CADR S))
    ((SUBRP (CAR S))
     (APPLY
       (CAR S)
       (MAAK-ARGUMENTLIJST (CDR S) A-LIJST)))
    (T
     (EVAL
       (DEFINITIE (CAR S))
       (MAAK-ASSOCIATIE-LIJST
         (DEFINITIE-ARGUMENTEN (CAR S))
         (CDR S)
         A-LIJST)))))
```

ASSOC zoekt een variabele op de a-lijst. MAAK-ARGUMENTLIJST maakt een nieuwe argumentlijst. MAAK-ASSOCIATIE-LIJST maakt een nieuwe associatielijst. Hier zijn de definities van deze functies.

De functie VOEG-BINDING-TOE voegt een binding toe aan een a-lijst:

```
(DEFINE VOEG-BINDING-TOE (VARIABELE BINDING A-LIJST)
  (CONS (LIST VARIABELE BINDING) A-LIJST))
```

Bijvoorbeeld (VOEG-BINDING-TOE 'A 'B NIL) ⇒ ((A B)) of gegeven de a-lijst A gelijk aan ((L (B C))) dan is de evaluatie van (VOEG-BINDING-TOE 'M '(D E)

A) gelijk aan ((M (D E)) (L (B C))).

De functie ASSOC zoekt een variabele op de a-lijst. Als er geen binding is, is het resultaat NIL. Bijvoorbeeld (ASSOC 'L '((M (D E)) (L (B C)))) is gelijk aan (B C).

```
(DEFINE ASSOC (VARIABELE A-LIJST)
  (COND
    ((NULL A-LIJST) NIL)
    ((EQ (CAAR A-LIJST) VARIABELE)
      (CADAR A-LIJST))
    (T (ASSOC VARIABELE (CDR A-LIJST)))))
```

De functie MAAK-ARGUMENTLIJST maakt van een argumentlijst een nieuwe argumentlijst waarbij elk argument geëvalueerd is. We kunnen de functie EVAL gebruiken om de evaluatie van een argument te verkrijgen. Bijvoorbeeld gegeven (MAAK-ARGUMENTLIJST '(L) '((L (A B)))) dan is het resultaat ((A B)) want de evaluatie van L is (A B). Hier is de definitie van MAAK-ARGUMENTLIJST:

```
(DEFINE MAAK-ARGUMENTLIJST (ARGUMENTLIJST A-LIJST)
  (COND
    ((NULL ARGUMENTLIJST) NIL)
    (T
     (CONS
       (EVAL (CAR ARGUMENTLIJST) A-LIJST)
       (MAAK-ARGUMENTLIJST
         (CDR ARGUMENTLIJST) A-LIJST)))))
```

Als de functie in de te evalueren LISP-object gelijk is aan een gedefinieerde functie, dan wordt EVAL recursief opgeroepen met een nieuwe a-lijst. De a-lijst bevat de bindingen tussen de argumenten in de functiedefinitie en het resultaat van de evaluatie van de corresponderende argumenten in het LISP-object. We hebben dus een hulpfunctie MAAK-ASSOCIATIELIJST nodig die gegeven een lijst van definitie-argumenten en een lijst van formule-argumenten, een nieuwe associatie-lijst oplevert. Bijvoorbeeld gegeven de formule (LENGTH '(A B C)) en de lijst van definitie-argumenten (L) uit

```
(DEFINE LENGTH (L)
  (COND
    ((NULL L) 0)
    (T (+ 1 (LENGTH (CDR L))))))
```

dan is de nieuwe associatie-lijst ((L (A B C))). Hier is de definitie:

```
(DEFINE MAAK-ASSOCIATIE-LIJST
        (DEFINITIE-ARGUMENTEN
         FORMULE-ARGUMENTEN
         A-LIJST)
  (COND
    ((NULL DEFINITIE-ARGUMENTEN) NIL)
    (T
      (VOEG-BINDING-TOE
        (CAR DEFINITIE-ARGUMENTEN)
        (EVAL (CAR FORMULE-ARGUMENTEN) A-LIJST)
        (MAAK-ASSOCIATIE-LIJST
          (CDR DEFINITIE-ARGUMENTEN)
          (CDR FORMULE-ARGUMENTEN)
          A-LIJST)))))
```

We veronderstellen ook een primitieve functie SUBRP die nagaat of een functie primitief is of gedefinieerd. Bijvoorbeeld, (SUBRP '+) is T omdat + een primitieve functie is.

En tenslotte veronderstellen we ook (DEFINITIE F) de definitie van een functie F oplevert en (DEFINITIE-ARGUMENTEN F) de argumenten van de functie F. Bijvoorbeeld voor de functie LENGTH is (DEFINITIE 'LENGTH) gelijk aan

```
(COND ((NULL L) 0)
      (T (+ 1 (LENGTH (CDR L)))))
```

en (DEFINITIE-ARGUMENTEN 'LENGTH) is gelijk aan (L).

2.1. OEFENINGEN

1. Construeer een verslag van EVAL en MAAK-ARGUMENTLIJST voor (+ (+ 5 10 6)).

```
1. ENTER EVAL
     S = (+ (+ 5. 3.) 9.)
     A-LIJST = NIL
  1. ENTER MAAK-ARGUMENTLIJST
       ARGUMENTLIJST = ((+ 5. 3.) 9.)
       A-LIJST = NIL
    2. ENTER EVAL
         S = (+ 5. 3.)
         A-LIJST = NIL
      2. ENTER MAAK-ARGUMENTLIJST
           ARGUMENTLIJST = (5. 3.)
```

```
            A-LIJST = NIL
          3. ENTER EVAL
              S = 5.
              A-LIJST = NIL
          3. EXIT EVAL 5.
          3. ENTER MAAK-ARGUMENTLIJST
              ARGUMENTLIJST = (3.)
              A-LIJST = NIL
            3. ENTER EVAL
                S = 3.
                A-LIJST = NIL
            3. EXIT EVAL 3.
            4. ENTER MAAK-ARGUMENTLIJST
                ARGUMENTLIJST = NIL
                A-LIJST = NIL
            4. EXIT MAAK-ARGUMENTLIJST NIL
          3. EXIT MAAK-ARGUMENTLIJST (3.)
        2. EXIT MAAK-ARGUMENTLIJST (5. 3.)
      2. EXIT EVAL 8.
      2. ENTER MAAK-ARGUMENTLIJST
          ARGUMENTLIJST = (9.)
          A-LIJST = NIL
        2. ENTER EVAL
            S = 9.
            A-LIJST = NIL
        2. EXIT EVAL 9.
        3. ENTER MAAK-ARGUMENTLIJST
            ARGUMENTLIJST = NIL
            A-LIJST = NIL
        3. EXIT MAAK-ARGUMENTLIJST NIL
      2. EXIT MAAK-ARGUMENTLIJST (9.)
    1. EXIT MAAK-ARGUMENTLIJST (8. 9.)
  1. EXIT EVAL 17.

  eindresultaat: 17.
```

2. De definitie van DERDE is

```
(DEFINE DERDE (X)
  (CAR (CDR (CDR X))))
```

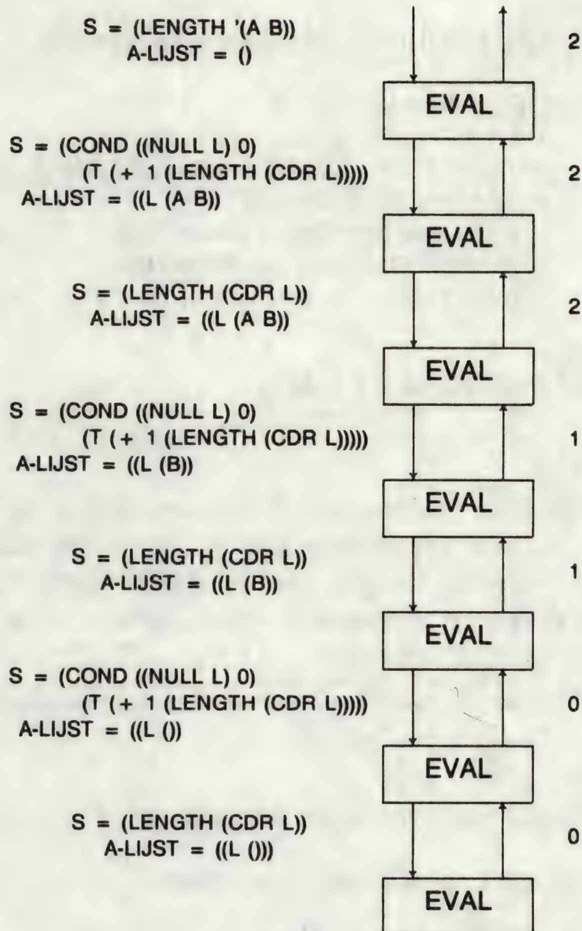
Construeer een verslag van (DERDE '(A B C)) voor de functies EVAL, MAAK-ASSOCIATIE-LIJST en MAAK-ARGUMENTLIJST.

```
1. ENTER EVAL
     S = (DERDE '(A B C))
     A-LIJST = NIL
  1. ENTER MAAK-ASSOCIATIE-LIJST
       DEFINITIE-ARGUMENTEN = (X)
       FORMULE-ARGUMENTEN = ('(A B C))
```

```
         A-LIJST = NIL
     2. ENTER EVAL
         S = '(A B C)
         A-LIJST = NIL
     2. EXIT EVAL (A B C)
     2. ENTER MAAK-ASSOCIATIE-LIJST
         DEFINITIE-ARGUMENTEN = NIL
         FORMULE-ARGUMENTE = NIL
         A-LIJST = NIL
     2. EXIT MAAK-ASSOCIATIE-LIJST NIL
   1. EXIT MAAK-ASSOCIATIE-LIJST (X (A B C))
   2. ENTER EVAL
       S = (CAR (CDR (CDR X)))
       A-LIJST = ((X (A B C)))
     1. ENTER MAAK-ARGUMENTLIJST
         ARGUMENTLIJST = ((CDR (CDR X)))
         A-LIJST = ((X (A B C)))
       3. ENTER EVAL
           S = (CDR (CDR X))
           A-LIJST = ((X (A B C)))
         2. ENTER MAAK-ARGUMENTLIJST
             ARGUMENTLIJST = ((CDR X))
             A-LIJST = ((X (A B C)))
           4. ENTER EVAL
               S = (CDR X)
               A-LIJST = ((X (A B C)))
             3. ENTER MAAK-ARGUMENTLIJST
                 S = (X)
                 A-LIJST = ((X (A B C)))
               5. ENTER EVAL
                   S = X
                   A-LIJST = ((X (A B C)))
               5. EXIT EVAL (A B C)
               4. ENTER MAAK-ARGUMENTLIJST
                   ARGUMENTLIJST = NIL
                   A-LIJST = ((X (A B C)))
                4. EXIT MAAK-ARGUMENTLIJST NIL
             3. EXIT MAAK-ARGUMENTLIJST ((A B C))
           4. EXIT EVAL (B C)
           3. ENTER MAAK-ARGUMENTLIJST
               ARGUMENTLIJST = NIL
               A-LIJST = ((X (A B C)))
           3. EXIT MAAK-ARGUMENTLIJST NIL
         2. EXIT MAAK-ARGUMENTLIJST ((B C))
       3. EXIT EVAL (C)
       2. ENTER MAAK-ARGUMENTLIJST
           ARGUMENTLIJST = NIL
           A-LIJST = ((X (A B C)))
       2. EXIT MAAK-ARGUMENTLIJST NIL
     1. EXIT MAAK-ARGUMENTLIJST ((C))
   2. EXIT EVAL C
 1. EXIT EVAL C
```

eindresultaat: C

3. Construeer een diagram voor de oproepen van LENGTH door EVAL voor (LENGTH '(A B)).



4. Verander de definities besproken uit dit hoofdstuk zodanig dat de evaluator een dynamisch bereik heeft.

Er verandert slechts één functie, namelijk MAAK-ASSOCIATIE-LIJST. De verandering zit hem in het feit dat de nieuwe bindingen *toegevoegd* moeten

worden aan de oude a-lijst. Dus in plaats van NIL krijgen we A-LIJST in de basisconditie van de recursie:

```
(DEFINE MAAK-ASSOCIATIE-LIJST
          (DEFINITIE-ARGUMENTEN
           FORMULE-ARGUMENTEN
           A-LIJST)
  (COND
    ((NULL DEFINITIE-ARGUMENTEN) A-LIJST)
    (T
     (VOEG-BINDING-TOE
       (CAR DEFINITIE-ARGUMENTEN)
       (EVAL (CAR FORMULE-ARGUMENTEN) A-LIJST)
       (MAAK-ASSOCIATIE-LIJST
          (CDR DEFINITIE-ARGUMENTEN)
          (CDR FORMULE-ARGUMENTEN)
          A-LIJST)))))
```

3. EEN REGELSYSTEEM IN LISP

3.1. INLEIDING

Het is niet moeilijk om in LISP zelf nieuwe talen te definiëren en daar evaluators voor te schrijven. Bij wijze van oefening gaan we nu een eenvoudig regelsysteem implementeren zoals dat tegenwoordig veel in expertsystemen voorkomt. Eerst beschrijven we de design. De ernstige lezer doet er goed aan om op basis van deze design zelf een implementatie te maken.

Een regelsysteem bestaat uit twee gedeelten: er is een verzameling feiten en een verzameling regels. De regels zijn van de vorm:

```
ALS
  een bepaald reeks van feiten het geval zijn
DAN
  kan men een bepaalde conclusie trekken
```

Gewoonlijk heeft de conclusie een bepaalde mate van zekerheid.

Bijvoorbeeld in het expertsysteem EMYCIN voor infectieziekten vinden we de regel

```
IF:
  1. The site of the culture is blood, and
  2. The patient has ecthyma gangrenosum skin lesions
THEN:
  There is strongly suggestive evidence [0.8] that the
  identity of the organism is pseudomonas.
```

In LISP vertaald kan een regel er als volgt uitzien:

```
(DEFREGEL NAAM-VAN-REGEL
  (ALS lijst van condities)
  (DAN conclusie zekerheid))
```

De condities en de conclusie bestaan uit symbolen die een bepaalde eigenschap beschrijven. Er is slechts één conclusie per regel. DEFREGEL is een gewone LISP-functie die bij executie de nodige interne datastructuren voor een regel maakt.

Hier is de vertaling van de EMYCIN regel:

```
(DEFREGEL
  (ALS SITE-OF-CULTURE-BLOOD ECTHYMA-GANGRENOSUM)
  (DAN ORGANISM-PSEUDOMONAS 0.8))
```

Of als we bijvoorbeeld regels maken voor diagnose van een auto dan zouden die er als volgt kunnnen uitzien:

```
(DEFREGEL BATTERIJ-LEEG
  (ALS HOOFDLICHTEN-ZWAK STARTER-DRAAIT-TRAAG)
  (DAN BATTERIJ-LEEG 0.7))
```

HOOFDLICHTEN-ZWAK, STARTER-DRAAIT-TRAAG en BATTERIJ-LEEG zijn eigenschappen. 0.7 is een graad van zekerheid.

Hier is nog een voorbeeld:

```
(DEFREGEL GEEN-BENZINE
   (ALS BENZINE-METER-OP-NUL AUTO-RIJDT-NIET)
   (DAN BENZINE-LEEG 0.9))
```

Eigenschappen worden geïntroduceerd met de syntaxis

```
(DEFEIGENSCHAP symbool)
```

zoals bijvoorbeeld in

```
(DEFEIGENSCHAP HOOFDLICHTEN-ZWAK)
```

DEFEIGENSCHAP is opnieuw een LISP-functie die de nodige interne datastructuren voor eigenschappen maakt.

We veronderstellen dat de gebruiker een vraag kan stellen over een bepaalde eigenschap met de functie IS?. Bijvoorbeeld de vraag: "is de benzine leeg?" komt overeen met:

(IS? BENZINE-LEEG)

Het systeem moet trachten hierop een antwoord te vinden aan de hand van de regels. Elke regel in het systeem die een bijdrage kan leveren voor het bepalen van de zekerheid van een eigenschap wordt getest. Dit kan ertoe leiden dat er nieuwe regels actief worden voor het testen van de condities. Eigenschappen waarvoor geen regels beschikbaar zijn worden onmiddellijk gevraagd aan de gebruiker die een graad van zekerheid aangeeft voor de betreffende eigenschap.

Hier is een voorbeeld van een kleine dialoog op basis van de gegeven regels:

(IS? BENZINE-LEEG)

Wat is de zekerheid van BENZINE-METER-OP-NUL
(geef getal tussen 1.0 en 0.0)? 1.0

Wat is de zekerheid van AUTO-RIJDT-NIET
(geef getal tussen 1.0 en 0.0)? 1.0

BENZINE-LEEG geldt met 0.9 zekerheid.

De implementatie valt uiteen in twee delen. Eerst ontwerpen we de diverse datastructuren. Dan bekijken we het deductiemechanisme.

3.2. DATASTRUCTUREN

De datastructuren kunnen met diverse technieken worden geïmplementeerd. We kiezen voor de techniek van abstracte datastructuren enkel gebaseerd op eigenschaplijsten om zeker te zijn dat vrijwel elke lezer de implementatie zelf kan proberen, zelfs al beschikt hij over een kleine computer. De implementatie met structuren is qua code eenvoudiger en is een goede oefening voor de geïnteresseerde lezer.

FEITEN

Een feit bestaat uit een eigenschap, en een zekerheid. Omdat de eigenschap een symbool is kunnen we de zekerheid er eenvoudigweg mee verbinden op de eigenschaplijst.

De volgende functies associëren een zekerheid met een eigenschap en zoeken deze opnieuw op:

```
(DEFMACRO DEFINIEER-ZEKERHEID (EIGENSCHAP ZEKERHEID)
  '(PUTPROP ,EIGENSCHAP ,ZEKERHEID 'ZEKERHEID))

(DEFMACRO HAAL-ZEKERHEID-EIGENSCHAP (EIGENSCHAP)
  '(GET ,EIGENSCHAP 'ZEKERHEID))
```

We associeëren ook met elke eigenschap de verzameling regels die kunnen bijdragen tot het bepalen van de zekerheid van deze eigenschap:

```
(DEFMACRO VOEG-REGEL-TOE (EIGENSCHAP REGEL)
  '(LET ((DE-EIGENSCHAP ,EIGENSCHAP))
     (PUTPROP DE-EIGENSCHAP
              (CONS ,REGEL (HAAL-REGELS DE-EIGENSCHAP))
              'REGELS)))

(DEFMACRO HAAL-REGELS (EIGENSCHAP)
  '(GET ,EIGENSCHAP 'REGELS))
```

Merk op dat we DE-EIGENSCHAP als nieuwe variabele invoeren om dubbele evaluatie van eigenschap te voorkomen.

De functie DEFEIGENSCHAP initialiseert een eigenschap. Dit wil zeggen dat de zekerheid gelijk gesteld wordt met ONBEKEND en dat de verzameling geassocieerde regels leeg is:

```
(DEFMACRO DEFEIGENSCHAP (EIGENSCHAP)
  '(PROGN ()
     (PUTPROP ',EIGENSCHAP NIL 'REGELS)
     (PUTPROP ',EIGENSCHAP 'ONBEKEND 'ZEKERHEID)))
```

REGELS

Regels hebben een conditioneel gedeelte, een conclusie en een zekerheid van de conclusie. De eigenschaplijst kan opnieuw gebruikt worden om deze eigenschappen bij te houden. Dus we definiëren opnieuw enkele eenvoudige macro's om de onderdelen van een regel te definiëren of op te zoeken:

```
(DEFMACRO DEFINIEER-REGEL
                  (REGEL-NAAM ALS-GEDEELTE
                   DAN-GEDEELTE ZEKERHEID)
  '(PROGN ()
     (PUTPROP ,REGEL-NAAM ,ALS-GEDEELTE 'ALS-GEDEELTE)
     (PUTPROP ,REGEL-NAAM ,DAN-GEDEELTE 'DAN-GEDEELTE)
     (PUTPROP ,REGEL-NAAM ,ZEKERHEID 'ZEKERHEID)))

(DEFMACRO HAAL-ALS-GEDEELTE (REGEL-NAAM)
  '(GET ,REGELNAAM 'ALS-GEDEELTE))

(DEFMACRO HAAL-DAN-GEDEELTE (REGEL-NAAM)
   '(GET ,REGELNAAM 'DAN-GEDEELTE))

(DEFMACRO HAAL-ZEKERHEID-REGEL (REGEL-NAAM)
   '(GET ,REGELNAAM 'ZEKERHEID))
```

De functie DEFREGEL moet alle onderdelen verbinden met de naam van de regel, en ook de regel toe voegen aan alle eigenschappen:

```
(DEFMACRO DEFREGEL (NAAM ALS-GEDEELTE DAN-GEDEELTE)
  '(LET
     ((REGEL-NAAM ',NAAM)
      (ALS-STUK (CDR ',ALS-GEDEELTE))
      (DAN-STUK (CDR ',DAN-GEDEELTE)))
     (DEFINIEER-REGEL REGEL-NAAM ALS-STUK
                      (CAR DAN-STUK)
                      (SECOND DAN-STUK))
     (MAPCAR #'(LAMBDA (CONDITIE)
                  (VOEG-REGEL-TOE REGEL-NAAM CONDITIE))
              ALS-STUK)
     REGEL-NAAM))
```

Merk op dat door de definitie van deze functies de interne representatie van regels en feiten volledig onafhankelijk wordt van het deductiesysteem of van de input van regels: we kunnen de interne representatie wijzigen zonder de implementatie van het deductiemechanisme te wijzigen, of zonder de vorm van de regels voor de gebruiker te wijzigen.

ZEKERHEDEN

Het is nodig om een calculus te ontwerpen voor het combineren van de zekerheid. Er zijn drie gevallen:

1. Het combineren van de zekerheden van de condities. Hiervoor gebruiken we de functie COMBINEER-VOOR-CONDITIE. Eén manier om dit te doen is door het minimum te nemen, op basis van het principe dat een redenering even sterk zal zijn als de zwakste schakel waarop ze gebouwd is.
2. Het combineren van de zekerheden van de conclusie van de regel met die van de condities. Hiervoor gebruiken we de functie COMBINEER-IN-REGEL. Er zijn allerlei oplossingen mogelijk. We zullen eenvoudigweg het gemiddelde nemen.
3. Het combineren van de resultaten van de diverse regels. Hiervoor definiëren we de functie COMBINEER-RESULTATEN. We gebruiken het maximum op basis van het principe dat als er meer dan één manier is om een feit af te leiden, de sterkste redenering bepaalt wat de sterkte is van de zekerheid van het feit.

We veronderstellen ook dat als één van de zekerheden die gecombineerd wordt ONBEKEND is, dat de andere zekerheid wordt genomen. Hier zijn dan de betreffende functies die de zekerheidscalculus implementeren:

```
(DEFUN COMBINEER-VOOR-CONDITIE (CF1 CF2)
   (COND ((ONBEKEND-P CF1) CF2)
         ((ONBEKEND-P CF2) CF1)
         (T (MINIMUM CF1 CF2))))

(DEFUN COMBINEER-IN-REGEL (CF1 CF2)
   (COND ((ONBEKEND-P CF1) CF2)
         ((ONBEKEND-P CF2) CF1)
         (T (GEMIDDELDE CF1 CF2))))

(DEFUN COMBINEER-RESULTATEN (CF1 CF2)
  (COND ((ONBEKEND-P CF1) CF2)
        ((ONBEKEND-P CF2) CF1)
        (T (MAXIMUM CF1 CF2))))
```

ONBEKEND-P gaat na of een zekerheid onbekend is:

```
(DEFMACRO ONBEKEND-P (CF)
   (EQUAL ,CF 'ONBEKEND))
```

3.3. HET DEDUCTIEMECHANISME

Het deductie algoritme ziet er als volgt uit:

Gegeven een te onderzoeken eigenschap P.

1. als de zekerheid van P bekend is, stop.
   als de zekerheid van P onbekend is dan
2. als er regels zijn om de zekerheid van P te berekenen, activeer de regels.
3. als er geen regels zijn vraag een zekerheid aan de gebruiker.

De activatie van regels verloopt als volgt.

1. Bereken de zekerheid van elk van de condities in de regels.
2. Combineer de zekerheden van de condities met de zekerheid van de conclusie en combineer dit geheel met de reeds bekomen zekerheid voor het desbetreffende feit.

We schrijven nu de LISP-functies nodig om dit algoritme te implementeren.

```
(DEFUN BEREKEN-ZEKERHEID (EIGENSCHAP)
  (LET
    ((BESTAANDE-ZEKERHEID
      (HAAL-ZEKERHEID-EIGENSCHAP EIGENSCHAP)))
    (IF (NOT (ONBEKEND-P EIGENSCHAP))
        BESTAANDE-ZEKERHEID
        (LET
          ((BESTAANDE-REGELS (HAAL-REGELS EIGENSCHAP)))
          (IF BESTAANDE-REGELS
              (ACTIVEER-REGELS
                BESTAANDE-REGELS EIGENSCHAP)
              (VRAAG-GEBRUIKER EIGENSCHAP))))))

(DEFUN ACTIVEER-REGELS (REGELS)
  (COND
    ((NULL REGELS) (HAAL-ZEKERHEID EIGENSCHAP))
    (T
      (ACTIVEER-EEN-REGEL (CAR REGELS) EIGENSCHAP)
      (ACTIVEER-REGELS (CDR REGELS)))))
```

```
(DEFUN ACTIVEER-EEN-REGEL (REGEL EIGENSCHAP)
  (DEFINIEER-ZEKERHEID EIGENSCHAP
    (COMBINEER-RESULTATEN
      (HAAL-ZEKERHEID EIGENSCHAP)
      (COMBINEER-IN-REGEL
        (HAAL-ZEKERHEID-REGEL REGEL)
        (APPLY 'COMBINEER-VOOR-CONDITIE
               (MAPCAR
                 #'(LAMBDA (CONDITIE)
                     (BEREKEN-ZEKERHEID CONDITIE))
                 (HAAL-CONDITIES REGEL)))))))

(DEFUN VRAAG-GEBRUIKER (EIGENSCHAP)
  (PRINT "Wat is de zekerheid van")
  (PRINT EIGENSCHAP)
  (PRINT "Geef getal tussen 1.0 en 0.0?")
  (LET ((ANSWER (READ)))
       (PUTPROP EIGENSCHAP ANSWER 'ZEKERHEID)))
```

Het geheel wordt opgeroepen vanuit de functie IS?:

```
(DEFMACRO IS? (EIGENSCHAP)
  `(BEREKEN-ZEKERHEID ,EIGENSCHAP))
```

UITBREIDINGEN

Hier zijn enkele kleine projecten om dit regelsysteem verder uit te bouwen.

1. Voeg een mechanisme toe dat kan uitleggen waarom een vraag gesteld wordt. Hint: introduceer twee stapels, een voor de gebruikte regels en een voor de reeds geteste condities.
2. Implementeer een voorwaarts regelsysteem. Telkens als een feit wordt toegevoegd zullen alle regels waarin de eigenschap voorkomt in het IF-gedeelte getest worden en eventueel hun conclusie afgeleid.
3. Implementeer een context-mechanisme, d.w.z. dat een eigenschap niet meer "globaal" is maar geassocieerd met een object.

4. SAMENVATTING

In dit gedeelte hebben we het evaluatiemechanisme grondiger bestudeerd door een eenvoudige evaluator te schrijven in LISP. De functies EVAL en APPLY zijn basisfuncties van LISP, zodanig dat de evaluator vanuit een programma expliciet

kan worden opgeroepen. We hebben ook een voorbeeld gezien hoe veranderingen in de evaluator leiden tot een andere semantiek.

Omdat het zo gemakkelijk is om evaluators te schrijven in LISP, wordt LISP veel gebruikt als basis voor nieuwe of hogere programmeertalen.

DEEL 7. LISP SYSTEMEN

1. LISP SYSTEMEN.
2. LISP DIALECTEN.
3. LISP MACHINES.
4. LISP CHIPS.
5. VERDERE ONTWIKKELINGEN.

1. LISP SYSTEMEN

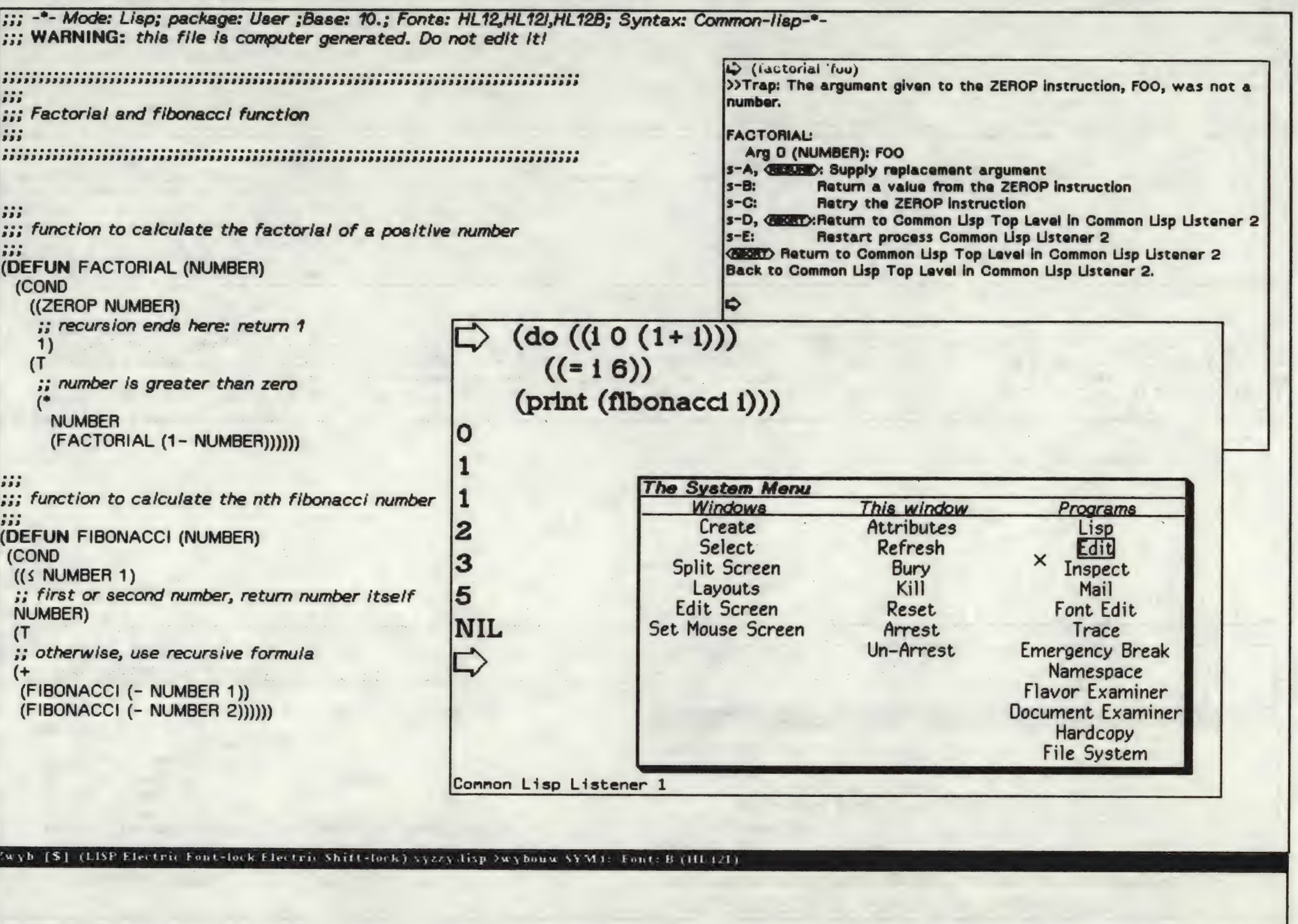
Een LISP systeem omvat meer dan een evaluator. Omdat LISP een volautomatisch geheugenbeheer heeft is er in de eerste plaats is een mechanisme nodig dat het geheugen beheert. Telkens als er lijstcellen nodig zijn worden die gelocaliseerd in het geheugen en als het geheugen volledig benut is, worden de cellen die niet meer nodig zijn opgespoord en terug vrij gemaakt. Dit proces heet 'garbage collection'.

Een LISP systeem bevat ook een compilator die functiedefinities omzet in machinetaal. Gecompileerde functiedefinities gedragen zich als primitieve functies. Zij kunnen dynamisch gecombineerd worden met andere functies en een functiedefinitie kan altijd terug veranderd worden.

Daarnaast bevat een LISP-systeem allerlei hulpmiddelen voor de programmeur, zoals een editor van lijststructuren, hulpmiddelen voor het opsporen van fouten (bijvoorbeeld maken van een verslag), enz.

Figuur 1. toont een typische schermlayout bij het schrijven van programma's op een interactief grafisch LISP-systeem. Er is een editing window waarin programma's geschreven worden. Dit window bevat de definities of de functies FACTORIAL and FIBONACCI. Rechts boven zien we een window waarin (FACTORIAL 'FOO) werd geëvalueerd. Merk op hoe het systeem reageert op een fout en hoe het mogelijk is om interactief verder te gaan. Onderaan links vinden we het systeem menu bovenop een ander LISP-window waar een eenvoudige LISP uitdrukking werd geëvalueerd.

Figuur 2 toont de schermlayout van een interactieve grafische debugger. Rechts bovenaan staat de functie die werd geëvalueerd. Het is mogelijk om diverse aspecten van de code te bekijken, zelfs de gecompileerde code in de linkerbovenhoek met een pijl naar het punt waar de evaluatie fout liep. Ook de bindingen van de variabelen en de evaluatiestack zijn zichtbaar en de gebruiker kan via de muis teruggegaan naar vroegere punten in de evaluatie.



Figuur 1: schermlayout bij programma ontwikkeling

```
TIMES
    14 PUSH-LOCAL FP|0          ;NUMBERS
    15 BUILTIN CDR STACK        ;creating NUMBERS(FP|1)
    16 PUSH-LOCAL FP|0          ;NUMBERS
    17 BUILTIN CAR STACK        ;creating ANS(FP|2)
    20 BRANCH 27
    21 PUSH-LOCAL FP|2          ;ANS
    22 PUSH-LOCAL FP|1          ;NUMBERS
    23 BUILTIN CAR STACK
=>  24 BUILTIN *-INTERNAL STACK
    25 POP-LOCAL FP|2           ;ANS
    26 BUILTIN CDR-LOCAL IGNORE FP|1    ;NUMBERS
```

More

```
DEFUN WEIGHTED-MEAN (FACTORS VALUES)
"function to calculate weighted mean of a series of numbers"
(/
 (APPLY #'+
        (MAPCAR
          #'(LAMBDA (FACTOR VALUE)
              (* FACTOR VALUE))
          FACTORS VALUES))
 (APPLY #'+ FACTORS)))
```

More

```
#<Stack-Frame TIMES PC=24>
```

```
Args:

Rest arg (NUMBERS): (5 A)
```

```
Locals:

Local 1 (NUMBERS): (A)
Local 2 (ANS): 5
```

More above

```
((LAMBDA (CL-USER::FACTOR CL-USER::VALUE) (* CL-USER::FACTOR CL-USER::VALUE)) (((VALUES (6 7 8 9 CL-USER::A)) (CL-USER::FACTORS (
(SYS:CALL-FUNNY-FUNCTION (SI:DIGESTED-LAMBDA (LAMBDA (CL-USER::FACTOR CL-USER::VALUE) (* CL-USER::FACTOR CL-USER::VALUE)) NIL 230
(SI:APPLY-LAMBDA (SI:DIGESTED-LAMBDA (LAMBDA (CL-USER::FACTOR CL-USER::VALUE) (* CL-USER::FACTOR CL-USER::VALUE)) NIL 2306 262658
(SI:*EVAL (* CL-USER::FACTOR CL-USER::VALUE) (((CL-USER::VALUE CL-USER::A) (CL-USER::FACTOR 5) (VALUES (6 7 8 9 CL-USER::A)) (CL-
→(TIMES 5 CL-USER::A)
```

More below

```
Return to normal debugger, staying in error context.
Supply replacement argument
Return a value from the *-INTERNAL instruction
Retry the *-INTERNAL instruction
Return to Common Lisp Top Level in Common Lisp Listener 2
Restart process Common Lisp Listener 2
```

What Error	Inspect	Return	Set arg	Retry	T
Arglist	Edit	Throw	Search		NIL

```
>>Trap: The second argument given to the ZETALISP-SYSTEM:*-INTERNAL instruction, A, was not a single-precision floating-point number
, a fixnum, or an extended number.
Type or mouse a function to edit, or mouse NIL to abort, or T for nothing:
T
```

Figuur 2: interactieve grafische debugger

2. LISP DIALECTEN

LISP is op het einde van de jaren '50 ontwikkeld onder de leiding van John McCarthy in het laboratorium voor Artificiële Intelligentie van het Massachusetts Institute of Technology. De taal was oorspronkelijk bedoeld als hulpmiddel voor het schrijven van programma's voor artificiële intelligentie, vandaar de nadruk op symbolische structuren, interactief programmeren en logische fundering. Deze opzet was zo succesrijk dat LISP nog altijd de universele taal voor artificiële intelligentie is.

Alhoewel LISP één van de eerste computertalen was, is het zich pas nu aan het verspreiden buiten de universiteiten, gedeeltelijk omdat toe-passingen van de artificiële intelligentie zoals natuurlijke taalverwerking en expert systemen praktisch bruikbaar geworden zijn.

De eerste implementatie van LISP werd voltooid in 1959 voor een IBM704. Sindsdien zijn er honderden implementaties gemaakt voor allerlei machines. De belangrijkste dialecten op dit moment zijn MACLISP en INTERLISP. MACLISP is ontwikkeld in het AI laboratorium van MIT. Hedendaagse belangrijke varianten van MACLISP zijn UCI-LISP, FRANZ-LISP, LISP Machine LISP (ook Zetalisp genoemd) en COMMON LISP. INTERLISP is ontwikkeld door Bolt, Beranek en Newman in Cambridge, Ma. en door het XEROX Palo Alto Research Center.

Elke implementatie bevat allerlei systemen die de programmeur helpen bij het ontwikkelen van programma's. INTERLISP heeft bijvoorbeeld een FILE PACKAGE dat boekhoudkundig werk verricht voor grote programma paketten, een interactief programma MASTERSCOPE voor analyse van INTERLISP programma's, een PROGRAMMER'S ASSISTANT die gegevens bijhoudt van wat een programmeur doet of gedaan heeft, enz.

De bibliografie bevat de belangrijkste referentiewerken voor MACLISP, INTERLISP en enkele andere belangrijke LISP dialecten.

Op dit moment is er een zeer belangrijke trend in de richting van COMMON LISP, het dialect dat ook in dit boek gebruikt werd. COMMON LISP is nu op zeer vele systemen beschikbaar, zelfs op persoonlijke computers, en evolueert langzaam tot de standaard voor LISP.

3. LISP MACHINES

LISP is een interactieve taal. Grote computers die hun tijd verdelen over verschillende gebruikers lenen zich minder tot interactief gebruik. In het midden van de jaren zeventig werd dit probleem ernstig en is men begonnen met

persoonlijke LISP machines te bouwen.

De LISP machines bevatten twee belangrijke innovaties:

1. LISP is de systeemtaal, d.w.z. dat alle systeemprogramma's, zoals editor, uitbatingssysteem, compilator, communicatie, tekstverwerking, grafisch werk, enz., in LISP geschreven zijn. Het belangrijkste voordeel hiervan is dat de gebruiker slechts één taal hoeft te kennen om alle aspecten van de computer te begrijpen. Hij kan bijvoorbeeld de editor uitbreiden door LISP functies te schrijven.
2. De volledige computer is onder de controle van één enkele gebruiker. Het belangrijkste voordeel hiervan is dat de gebruiker niet langer hoeft te wachten tot activiteiten van andere gebruikers voltooid zijn, dat hij het geheugen (ook extern geheugen) helemaal voor zich heeft, enz.

De bedoeling is dat elke onderzoeker een LISP-machine in zijn bureel heeft. De machines zijn verbonden met elkaar via een netwerk. Het netwerk heeft ook aftakkingen naar standaard computers (bijvoorbeeld om data op te slaan of te communiceren met gebruikers op deze computers), en naar printers en communicatielijnen naar buiten (telefoon, sateliet). De LISP machine wordt niet alleen gebruikt om te programmeren of om programma's uit te voeren, maar ook om teksten te schrijven, brieven te schrijven en te versturen, enz.

De eerste LISP machines werden geconstrueerd in het AI laboratorium van MIT in het midden van de jaren zeventig. Hieruit zijn twee bedrijven gegroeid die deze computers verkopen en verder ontwikkelen: Symbolics en LMI. Ondertussen zijn er ook andere bedrijven (o.m. XEROX) die LISP machines produceren, vooral in Japan. Op dit moment zijn er reeds honderden machines in gebruik in de Verenigde Staten. Bij wijze van voorbeeld volgt hier een korte beschrijving van de SYMBOLICS 3600.

Deze machine bevat een uitgebreid gamma programmeringshulpmiddelen, zoals een systeem voor beheer van het scherm, een krachtige editor, een incrementele compilator en dynamische linker, een systeem voor analyse van fouten dat gebruik maakt van het scherm en een systeem voor organisatie en manipulatie van extern geheugen en netwerk faciliteiten. De machine zit in een kast van ongeveer 75 cm hoog en heeft geen speciale vereisten wat betreft temperatuur of vochtigheidsgraad.

De hardware is gebaseerd op een 36 bit processor speciaal ontwikkeld voor symbolische verwerking. De processor heeft 32 bit datapaden. Hij kan op microniveau geprogrammeerd worden. De processor doet run-time checking voor fouten in datatype, ongeïnitialiseerde variabelen, enz. Het voert ongeveer 1 miljoen LISP

instructies uit per seconde.

De machine heeft 1.125 Mbyte intern geheugen en een 169 Mbyte "Winchester" schijf extern geheugen met snelle toegang. Er is een interface met een 10 Mbit/sec Ethernet voor communicatie met andere computers of met periferen. Er is een zwart/wit scherm van 1000 lijnen of een 1000 x 1000 kleurscherm. De bitmap voor het scherm is geïntegreerd in het intern geheugen. Er is een uitgebreid toetsenbord en een muis waarmee een punt op het scherm geaddresseerd kan worden.

4. LISP CHIPS

VLSI (Very Large Scale Integration) maakt het mogelijk om vrij complexe systemen op één enkele chip te implementeren. Het duurde dan ook niet lang voor pogingen werden ondernomen om een LISP interpreteerder op een chip te zetten. Een voorbeeld is een LISP chip geconstrueerd onder leiding van G. Sussman in het MIT laboratorium voor Artificiële Intelligentie.

Op deze LISP chip zijn lijststructuren gerepresenteerd in cellen. Een cel bevat een 32-bit gedeelte voor de CAR van een lijst en een 32-bit gedeelte voor de CDR van een lijst. Elke gedeelte bevat niet alleen een datum maar ook een 7-bit indicator die aangeeft wat het datum element is, bijvoorbeeld een lijst, een symbool, enz., en 1 bit voor de 'garbage collector' die aangeeft of de cel gebruikt wordt of niet. Data, programma's, en zelfs het schijfregister zijn gerepresenteerd in de vorm van deze lijststructuren.

De architectuur zelf is een standaard von Neumann architectuur: een processor verbonden met een geheugen. De processor bevat datapaden en een controle-eenheid. De datapaden bevatten een reeks specifieke registers met ingebouwde operatoren. Er is bijvoorbeeld een register voor de waarde van de laatste uitdrukking, een register voor de s-uitdrukking die op dit moment wordt geëvalueerd, enz. De registers zijn verbonden met een 32-bit bus. De controle eenheid is een eindige automaat die programma's in microcode doorloopt. Een van deze programma's is de evaluator. Operaties zoals CAR en CDR zijn primitieve machine operaties. De chip bevat ook een 'garbage collector', een programma dat ervoor zorgt dat LISP cellen beschikbaar zijn wanneer nodig en dat LISP cellen die niet langer meer nodig zijn opnieuw bruikbaar worden.

5. VERDERE ONTWIKKELINGEN

Alhoewel de basis van LISP zelf constant blijft, is er een intense ontwikkeling aan de gang. Ten eerste wordt er druk geëxperimenteerd met nieuwe types van evaluatoren, zoals 'luie' evaluators die evaluatie uit-stellen tot er behoefte is aan het resultaat, partiële evaluators die gedeeltelijke evaluatie verrichten vóór de eigenlijke evaluatie begint, enz...

Ten tweede wordt er onderzoek verricht hoe nieuwe programmeerstijlen in LISP mogelijk zijn. Een voorbeeld hiervan is objectgericht programmeren: definities van functies zijn niet langer globaal maar specifiek voor de objecten waarop de functie toegepast wordt en een object verkrijgt sommige van zijn definities door een ervingsmechanisme. Een ander voorbeeld zijn de diverse pogingen om logisch programmeren à la PROLOG in LISP te realiseren.

Een derde gebied van onderzoek vormen de LISP machines en LISP chips. Er wordt onder meer gewerkt aan nieuwe non-von Neumann computerarchitecturen die parallelle evaluatie van LISP kunnen uitvoeren. Een voorbeeld hiervan zijn de reductiemachines.

Een vierde gebied vormen parallelle LISP implementaties. Door de mogelijkheden geboden door VLSI evolueert de informatica naar parallelle computers. Het is dan ook niet verwonderlijk dat er onderzoek wordt gedaan op diverse plaatsen naar parallelle LISP systemen. Een voorbeeld hiervan is *LISP (Hillis en Steele, 1986) dat operationeel is op de Connection Machine (Hillis, 1985). De Connection Machine bevat 64.000 echte processoren en kan hiermee virtueel miljoenen processoren simuleren. Een LISP-machine dient als front-end.

Tenslotte is er intens onderzoek in programmeringshulpmiddelen, mede door de beschikbaarheid van persoonlijke LISP machines die een groot scherm hebben en een manier om direct naar een punt op het scherm te wijzen. Hierdoor wordt het mogelijk om via menuselectie te programmeren, om in verschillende onderdelen van het scherm aspecten van het evaluatieproces te laten verschijnen, enz.

Deze vitaliteit van LISP is merkwaardig en het ziet er niet naar uit dat ze zal afnemen.

GEANNOTEERDE BIBLIOGRAFIE

ALLEN, J.

The Anatomy of LISP. Mc-Graw Hill Book Cy, New York, 1978.

Een inleiding in fundamentele concepten van de informatica via LISP. Sterk aangeraden alhoewel LISP 1.5. notatie wordt gebruikt. Gaat ook in op de interne werking van LISP, o.a. de LISP compiler.

BACKUS, J.

Can Programming be liberated from the Von Neumann style? A functional style and its algebra of programs. Comm ACM. vol 21, nr 8, p. 613-641.

Beschrijft een applicatieve taal gebaseerd op combinatorische logica in de plaats van lambda-calculus.

BERKELEY, E. en D. BOBROW (eds.)

The programming language LISP: Its operation and Applications, MIT Press, 1966. Cambridge, Mass, 1964.

Een van de eerste verzamelingen teksten over LISP. Bevat een goede korte inleiding door Berkeley.

BOYER, R. en J. MOORE.

Proving theorems about LISP functions. Jour ACM, 1, p. 129-144, Jan 1975.

Bevat voorbeelden van het bewijzen van eigenschappen van LISP functies.

BURGE, W. H.

Recursive Programming Techniques. Addison-Wesley Publishing Company. Reading, Massachussetts.

Bevat programmeertechnieken voor talen gebaseerd op de lambda-calculus.

BURTON, R., et.al.

Papers on Interlisp-D. XEROX PARC report, SSL-80-4, Palo Alto, Calif., 1980.

Bevat meer informatie over de DOLPHIN.

CHARNIAK, E. C. RIESBECK en D. McDERMOTT

Artificial Intelligence Programming, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Behandelt technieken van LISP programmatie voor artificiële intelligentie.

CHURCH, A.

The Calculi of Lambda-conversion. Annals of Mathematics Studies, Princeton University Press, New Jersey, 1941.

Bevat de logische grondslagen van LISP.

DARLINGTON, J., P. HENDERSON and D.A. TURNER

Functional Programming and Its Applications. An advanced course. Cambridge University Press, Cambridge, 1982.

Geeft een goed overzicht van recente ontwikkelingen in functionele talen, o.m. ook semantiek en nieuwe hardware voor functionele talen.

FODERARO, J.

The FRANZ LISP Manual. University of California, Berkeley California, 1979.

De standaard referentie voor FRANZ LISP, een MACLISP implementatie voor de VAX.

FRIEDMAN, D.

The Little LISPer. SRA, Menlo Park, 1974.

Een leuke inleiding in puur LISP via oefeningen.

GREENBERG, B.

Notes on the Programming Language LISP. Student Information Processing Board, MIT, Cambridge Mass.

Een uitstekende inleiding in LISP.

GREENBLATT, R.

The LISP Machine. MIT AI lab. Working Paper 79, MIT, Cambridge Mass, 1974.

Het oorspronkelijke voorstel voor een LISP machine.

HENDERSON, P.

Functional Programming. Application and Implementation. Prentice-Hall, Englewood Cliffs, New Jersey.

Bevat niet alleen een basis van applicatief programmeren maar ook onderwerpen zoals machine architectuur, uitgestelde evaluatie, parallellism in applicatief programmeren, functies van hogere orde, enz. De taal is een variant van LISP.

HILLIS, D.

The Connection Machine, MIT Press, Cambridge Mass, 1986.

Bevat een beschrijving van de Connection Machine en een parallelle LISP implementatie.

KNIGHT, T.

The CONS Microprocessor, MIT AI lab. Working paper 80, MIT, Cambridge Mass, 1974.

Beschrijving van de MIT LISP machine hardware.

MAURER, W. D.

A Programmer's Introduction to LISP. American Elsevier, New York, 1973.

McCARTHY, J.

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Comm. ACM, p. 184-195, 1960.

Het artikel waarin McCarthy voor het eerst LISP voorstelde.

McCARTHY, J. et.al.

LISP 1.5 Programmer's Manual, MIT Press, Cambridge Mass., 1966.

De manual van het eerste LISP systeem. Nu vooral van historisch belang.

McCARTHY, J.

History of LISP, SIGPLAN Notices, Vol. 13, no.8, August 1978, pp. 217-223.

MEEHAN, J.

The New UCI LISP Manual, Lawrence Erlbaum Ass. Hillsdale, New Jersey, 1979.

De referentie voor UCILISP, een belangrijke variant van MACLISP.

MOON, D.

The MACLISP manual. MIT AI Lab, Cambridge Mass, 1976.

De standaard referentie voor MACLISP.

MOORE, J.

The Interlisp Virtual Machine Specification. XEROX PARC Report. CSL-76-5. Palo Alto, Calif., 1976. (tweede versie 1979).

Bevat de definitie van het Interlisp systeem op de DOLPHIN.

REES, J. and N. ADAMS

T, a dialect of LISP. pag. 114-122 in Proceedings of the 1980 ACM Conference on LISP and Functional Programming. Pittsburgh, 1980.

Een nieuwe LISP voor de MC68000, geïnspireerd op NIL en SCHEME.

ROBINSON, J. A. en I. SIBERT

LOGLISP: Motivation, design and implementation. p. 299-313. in Clark, K.L. and S. Tarnlund (eds.) Logic Programming. Academic Press, London, 1982.

Beschrijft een voorbeeld van uitbreidingen van LISP met constructies om logisch te programmeren.

SIKLOSSY, L.

Let's Talk LISP, Prentice Hall, Englewood Cliffs, New Jersey, 1976.

Inleiding in LISP.

STEELE, G.

An Overview of Common LISP. p. 98-107 in ACM Conference Record on LISP and Functional Programming. Pittsburgh, 1982.

Bevat de motivatie en een beschrijving van COMMON LISP, een nieuw LISP dialect dat bedoeld is als standaard voor toekomstige LISP systemen.

STEELE, G.

Common Lisp: The Language, Digital Press, Burlington, 1984.

Hèt referentiewerk inzake COMMON LISP.

STEELS, L.

ORBIT: An applicative view of object-oriented programming. in: Degano, P. and E. Sandewall. Integrated Interactive Computer Systems. North-Holland, Amsterdam, 1982.

Bevat een voorbeeld van constructies voor objectgericht programmeren in LISP.

STOYAN, H.

LISP - Anwendungsgebiete, Grundbegriffe, Geschichte. Akademie-Verlag, Berlin, 1980.

Bevat een geschiedenis van LISP en heel wat technische details over de belangrijkste LISP dialecten.

SUSSMAN, G.

LISP, Programming and Implementation. in Darlington, et.al. (1982), p. 29-71.

Uitstekende korte inleiding in LISP en de implementatie van LISP.

SUSSMAN, G., J. HOLLOWAY, G. STEELE, en A. BELL.

SCHEME-79 - Lisp on a Chip. IEEE Computer Architecture, July 1981.

Beschrijving van een variant van Lisp op een chip.

TEITELMAN, W.

INTERLISP Reference Manual. XEROX Palo Alto Research Center. Palo Alto, Calif., 1974-78.

De standaard referentie voor INTERLISP.

TEITELMAN, W. and L. MASINTER.

The INTERLISP Programming Environment. IEEE Computer April 1981, pp.25-33.

Een overzicht van de programmeermiddelen voor INTERLISP.

WEINREB, D. en D. MOON

LISP Machine Manual. MIT AI Lab. MIT, Cambridge Mass. 1981.

De standaard referentie voor LISP Machine LISP.

WEISSMAN, C.

LISP 1.5 Primer, Dickenson Press, 1967

Een van de allereerste handboeken van LISP. Nog altijd de moeite waarde.

WINSTON, P. en B. HORN

LISP, Addison-Wesley Pub. Cy. Reading, Mass. 1981.

Uitstekende inleiding in LISP. Het tweede deel is een inleiding in artificiële intelligentie.

INDEX

1. OVERZICHT VAN SPECIALE TEKENS

2. INDEX

ACADEMIC SERVICE INFORMATICA UITGAVEN

AUTOMATISERING EN COMPUTERS

Computers en onze informatiemaatschappij - M.A. Arbib
Computers in de negentiger jaren - G.L. Simons
De informatiemaatschappij - Jan Everink
Op weg naar een risicoloze maatschappij? - Jan Holvast
Basiskennis informatieverwerking - Jan Everink
AIV, Automatisering van de informatieverzorging - Th.J.G. Derksen & H.W. Crins
BIV, Basis van de geautomatiseerde informatieverzorging - Th.J.G. Derksen & H.W. Crins
Organisatie, informatie en computers - David M. Kroenke
Computers in de basisschool - H. Lamers & J.A. Wegkamp
Effectieve toepassingen van computers - M. Peltu

MICROCOMPUTERS

Microcomputers thuis en op school - K.P. Goldberg & D. Sherwood
Bouw zelf een expertsysteem in BASIC - Chris Naylor
Programmeercursus MicrosoftBASIC - Nok van Veen
Werken met bestanden in BASIC - LeRoy Finkel & Jerald R. Brown
Programmeercursus BASIC op de Commodore 64 - Nok van Veen
Werken met bestanden op de Commodore 64 - G. Fisher, L. Finkel & J.R. Brown
Het Electron en BBC Micro boek - Jim McGregor & Alan Watt
Werken met bestanden op de Apple - LeRoy Finkel & Jerald R. Brown
Programmeercursus ApplesoftBASIC - Nok van Veen & Ad van Delft
Programmeercursus MSX BASIC - Nok van Veen
Werken met bestanden in MSX BASIC - LeRoy Finkel & Jerald R. Brown
40 grafische programma's - voor de Commodore 64; voor de Electron en BBC; voor de ZX Spectrum; voor de Apple II, IIe en IIc; in MSX BASIC - Marcel Sutter

MICROPROCESSORS EN ASSEMBLEERTALEN

Procescomputers, basisbegrippen - J.E. Rooda & W.C. Boot
Cursus Z-80 assembleertaal - Roger Hutty
6502 assembleertaal en machinecode voor beginners - A.P. Stephenson
EXAT-handboek - Micro-Teach

BESTURINGSSYSTEMEN

Inleiding besturingssystemen - A.M. Lister
Theorie en praktijk van besturingssystemen - J.L. Peterson & A. Silberschatz
Systeemprogrammatuur en softwareontwikkeling voor microcomputers - E. Verhulst
Bedrijfssystemen - EIT-serie, deel 4
CP/M, het operating system voor microcomputers - J.N. Fernandez & R. Ashley
CP/M 86, een besturingssysteem voor 16 bit microcomputers - J.N. Fernandez & R. Ashley
CP/M voor gevorderden - A. Clarke e.a.
PC DOS, het besturingssysteem van de IBM PC - R. Ashley & J.N. Fernandez
MS DOS, het besturingssysteem voor 16 bit microcomputers - R. Ashley & J.N. Fernandez
UNIX, het standaard operating system - G.J.M. Austen & H.J. Thomassen
De UNIX programmeeromgeving - B.W. Kernighan & R. Pike

PERSONAL COMPUTERS EN PROGRAMMAPAKKETTEN

De IBM PC en zijn toepassingen - Laurence Press
Werken met bestanden in IBM- en GW-BASIC - J.R. Brown & LeRoy Finkel
40 grafische programma's in IBM- en GW-BASIC - Marcel Sutter
Werken met VisiCalc - C. Klitzner & M.J. Plociak Jr.
Multiplan, een hulpmiddel bij de bedrijfsvoering - D.F. Cobb e.a.
Werken met Lotus 1-2-3 - G.T. LeBlond & D.F. Cobb
Lotus 1-2-3: Tips, Trucs en Tegenvallers - D. Andersen & D.F. Cobb
Lotus 1-2-3: Financiële macro's - Thomas W. Carlton
Symphony deel I en II - D.P. Ewing & G.T. LeBlond
dBASE III handboek - George Tsu-der Chou
WordStar stap voor stap - Ruth Ashley & Judi N. Fernandez

PROGRAMMEREN

Een methode van programmeren - Edsger W. Dijkstra & W.H.J. Feijen
Programmeren, met ontwerpen van algoritmen (met Pascal) - J.J. van Amstel
Voortgezet programmeren, het ontwerpen van datastructuren en algoritmen - J.J. van Amstel & J.A.A.M. Poirters

Problemen oplossen met de computer - R.G. Dromey
Inleiding tot het programmeren, deel 1 en 2 - J.J. van Amstel e.a.
Het Groot Pascal Spreukenboek - Henry F. Ledgard e.a.
Software engineering, het bouwen van grote programma's - I. Sommerville
JSP Jackson struktureel programmeren - Henk Jansen
JSP Uitwerkingenboek, JSP Procedureboek - Henk Jansen

PROGRAMMEERTALEN

Aspecten van programmeertalen - J.J. van Amstel & J.A.A.M. Poirters
Programmeertalen, een inleiding - J.J. van Amstel e.a.
Colloquium programmeertalen - red. J.A.A.M. Poirters & G.J. Schoenmaker
BASIC - EIT-serie, deel 3
Cursus BASIC, een practicum handleiding voor BASIC op de PRIME - R. Bloothoofd e.a.
Een programmeercursus in BASIC - Nok van Veen & René Wissing
Cursus Pascal - A. van der Sluis & C.A.C. Görts
Cursus eenvoudig Pascal - A. van der Sluis & C.A.C. Görts
Inleiding programmeren in Pascal - C. van de Wijgaart
Modula-2 - E. Verhulst
Systeemontwikkeling met Ada - Grady Booch
Cursus COBOL - A. Parkin
Cursus FORTRAN 77 - J.N.P. Hume & R.C. Holt
De programmeertaal C - L. Ammeraal
Flitsend Forth - Alan Winfield
Logisch LOGO - Auke Sikma
Programmeren in LISP - L.L. Steels
Micro-PROLOG, programmeren in logica - K.L. Clarke & F.G.McGabe
Inleiding PROLOG - W. Burnham & A. Hall

BESTANDSORGANISATIE, DATABASE EN GEGEVENSANALYSE

Bestandsorganisatie - R.J. Lunbeck & F. Remmen
Database, een inleiding - C.J. Date
Databases, grondslagen voor de logische structuur - F. Remmen
SQL in de praktijk - H.B. Eilers e.a.
Het SQL Leerboek - Rick F. van der Lans
Gegevensanalyse - R.P. Langerhorst

INFORMATIE-ANALYSE EN SYSTEEMONTWERP

Inleiding systeemanalyse en systeemontwerp - W.S. Davis
Systeemontwikkeling zonder zorgen - Paul T. Ward
Systeemontwikkeling volgens SDM - H.B. Eilers
Samenvatting SDM - Pandata
Systeemontwikkeling volgens JSD - Michael Jackson
Informatie-analyse volgens NIAM - J.J.V.R. Wintraecken
Information engineering - J. Blank
Het ontwerpen van interactieve toepassingen en computernetwerken - J.A. Scheltens
EDP Audit - C. de Backer
Management informatiesystemen - G.B. Davis & M.H. Olson
Prototyping, een instrument voor systeemontwerpers - red. T. Hoenderkamp & H.G. Sol
Het ontwikkelen van informatiesystemen met prototyping - R. Vonk
Simulatie, een moderne methode van onderzoek - S.K.T. Boersma & T. Hoenderkamp

EXPERTSYSTEMEN EN KUNSTMATIGE INTELLIGENTIE

Kunstmatige intelligentie - Patrick H. Winston
Expertsystemen - Henk de Swaan Arons & Peter van Lith
Ontwikkelingen in expertsystemen - red. A. Nijholt & L.L. Steels

THEORETISCHE INFORMATICA EN SYSTEEMPROGRAMMATUUR

Systeemprogrammatuur - H. Alblas
Vertalerbouw - H. Alblas e.a.

OPERATIONELE RESEARCH

Operationele research - Y.M.I. Dirickx e.a.
Lineaire programmering als hulpmiddel bij de besluitvorming - S.W. Douma



OVER DE AUTEUR:

LUC STEELS studeerde informatica aan het Massachusetts Institute of Technology na een doctoraat over computerlinguïstiek aan de universiteit van Antwerpen (UIA). Hij was werkzaam als wetenschappelijk medewerker in het laboratorium voor Kunstmatige Intelligentie van M.I.T. Daarna werd hij programmeerleider voor geologische expertsystemen in de laboratoria van Schlumberger in de Verenigde Staten. Momenteel doceert Steels Informatica en Artificiële Intelligentie aan de Vrije Universiteit van Brussel, en leidt het laboratorium voor Artificiële Intelligentie van de VUB.

OVER HET BOEK:

PROGRAMMEREN IN LISP is in de eerste plaats een inleiding in de programmeertaal LISP. Het behandelt uitvoerig de belangrijkste primitieve datastructuren van LISP: atomen, lijsten, getallen, strings, eigenschaplijsten en arrays, de verschillende typen van functies, de diverse mechanismen voor de definitie van nieuwe functies, en het evaluatieproces. Het boek is echter ook een inleiding in het programmeren in LISP met de nadruk op het exploreren van diverse stijlen. De belangrijkste applicatieve definitieschema's, voornamelijk van het recursieve type, worden bestudeerd. Het boek behandelt hoe functies van hogere order kunnen worden gedefinieerd en hoe het daarom mogelijk is om een functionele stijl te beoefenen in LISP. Het behandelt ook methoden voor de constructie van abstracte datastructuren en toont aan hoe men kan objectgericht programmeren in LISP.

PROGRAMMEREN IN LISP kan dienen als handboek bij een cursus over LISP of voor zelfstudie. Na lectuur moet de lezer in staat zijn om LISP-programma's te schrijven en om programma's van anderen te lezen en te begrijpen. Het boek is opgezet in een interactieve stijl: Elk hoofdstuk introduceert enkele concepten aan de hand van definities en voorbeelden. Daarna komen een tiental opgeloste oefeningen. Op het einde van elk deel komen dan nog enkele gemengde opgeloste opgaven van grotere omvang.

Deze tweede druk is volledig gebaseerd op COMMON LISP. Ook werden nieuwe oefeningen toegevoegd en nieuwe thema's zoals closures, packages, en de implementatie van een regelsysteem behandeld.

VOORKANT: Zicht op de console van een Symbolics 3600 LISP machine